MAYA®

Toward the Universal Database: U-forms and the VIA Repository

Doc. No. MTR–02001

Peter Lucas
lucas@maya.com

Jeff Senn
senn@maya.com

MAYA Design, Inc.
2100 Wharton Street
Pittsburgh, PA 15203

T: 412-488-2900
F: 412-488-2940
maya@maya.com
www.maya.com

"Achieving some basic degree of understanding among different computers to make automatization possible is not as technically difficult as it sounds. However, it does require one very difficult commodity: human consensus."

—Michael Dertouzos

For a decade now, researchers have been talking about the emergence of "One Huge Computer," that is, an architecture that permits the Internet and its billions of attached computing devices to essentially function as a single, albeit massively distributed and decentralized, information system. This provocative vision has motivated many of the most important developments in contemporary computer science. Mobile code, device discovery protocols, N-tier and peer-to-peer architectures, and many other advances were conceived in pursuit of this vision. We have a long way to go in achieving that vision, and there are many difficult problems remaining to be solved, but the first steps on the path to the Universal Computer now seems fairly clear.

Far less clear, however, is the path toward another, equally provocative and challenging goal: that of the Universal Database. If we propose to build a computer that spans the planet, will we not also need an architecture for the persistent storage and organization of data that is capable of scaling-both in size and in space-to a similar degree? How can we simultaneously support the requirements for rapid, flexible local access, distributed implementations and administration, stability, evolvability, security and all of the other imperatives that such a database would entail?

Such a system will not be built by assembling federations of relational databases interconnected via transactional links - the laws of physics suffice to guarantee that. Just as the needs of distributed, network-centric computing have demanded new design patterns, tools, and abstractions, creating the Universal Database will similarly require an architectural discontinuity from current practice. This paper describes the beginnings of such an architecture, and a component-the VIA Repository-designed in accordance with that architecture.

Requirements

All large-scale progress in the evolution of computing has been driven by the development and widespread adoption of simple, general abstractions. Digital computing itself is founded on an abstract data type, namely the bit. Processor and peripheral architectures were rationalized in the 1970s by the universal adoption of another abstract data type: the byte. Similarly, the Internet was developed around the idea of the packet.

In the world of database systems, the prevailing abstraction is that of the relational table. The development of this abstraction, along with the intimately related Structured Query Language (SQL) provided the foundation upon which the modern database industry is based. Understanding why this abstraction is inappropriate as a foundation for the Universal Database requires us to examine the requirements that such an abstraction must fulfil. Foremost among these requirements are these three: data liquidity, object extensibility, and universal identity.

Data Liquidity

It is obvious that the Universal Database must be vastly distributed, both in space and across many diverse devices. This stipulation implies the need for a high level of data liquidity, that is, the ability for data to flow readily from venue to venue. Each computing device and each physical location represents such a venue. Since no single venue could possibly hope to accumulate more than a tiny fraction of the entire corpus locally, it is clear that the operation of the Universal Database will entail the frequent and routine spatial relocation of data as a fundamental aspect of its operation. Furthermore, if the Universal Database is to scale downward to small, mobile computing devices such as Palm Pilots and cell phones, the currency of this flow must not be too molar; that is, data must be separable into small modular units that can be relocated independently of the larger collections of which they are a part.

What is the basic currency of a relational database? For many purposes it would be desirable to say that it is the table entry or “row” of a table. This is not quite the case, however. One cannot really separate a row of a table from the table itself-the identity of the two are inextricably bound together. This is true for two reasons: (1) In the relational model, the structure of each entry (and therefore one's ability to properly interpret that entry) is not a property of the entry itself, but rather a property of the table of which the entry is a member. That is, the schema by which one may interpret the data stored in the entry is intrinsic to the table, not the item. Moreover, the schema of an item cannot be changed without simultaneously changing the schemata of all other items in the table. (2) The definitional identity of each row (to the extent that it is well-defined at all) is tied to the table's so-called “primary key,” a column, or combination of columns whose value is guaranteed to be unique within the table. In essence, the value of an entry's primary key defines the identity of that item. Although this value is intrinsic to the entry rather than the table, the uniqueness (and therefore the identity) of the row is scoped only to the table, and cannot be assured if the row is removed from the table. For both of these reasons, an entry of a relational table is fundamentally inseparable from the table itself. Thus, the unit of currency in the relational model must be considered to be the table, not the row. This is not to say, of course, that the data contained in a row cannot be extracted from the table and moved to another table via some serialization protocol involving some kind of markup or other external encoding of the data. Such an operation, however, requires quite a bit of coordination and prior agreement between the two venues, and in any event involves services external from the databases architecture, rather than being a capability of a distributed database per se.

Object Extensibility

The Universal Database will contains trillions of instances of billions of different kinds of data objects. These objects will vary widely in the precision with which their content is defined and also in how widely those definitions will be recognized as authoritative. This is very similar to the structuring of information in the world of paper-based data: There exist many diverse paper forms, ranging from laboriously-designed IRS tax forms used by millions, to casually produced "sticky-pad" message forms used for informal intra-office communications. In addition, of course, a great deal of information is captured and transmitted on blank sheets of paper, relying on semiformal conventions and common sense for its interpretation. It is precisely this range of formality and the extensibility afforded by blank pages (and scribbles in margins of more highly structured forms) that allowed the single medium of paper and ink to serve successfully for so many centuries as a universal communications/storage medium.

As we have already seen, computer databases are very different. The standard models all require a priori specifications of storage schemata during the design stage of the system in question, and changes to those specifications are heavyweight, non-routine operations. It is as if we were to attempt to run a paper-based economy, relying only on an exhaustive inventory of pre-printed forms, without first making available blank sheets of paper. Such prior specification of the detailed structure of the Universal Database seems clearly out of the question. Rather, the fundamental unit of data will, we argue, need to be by its basic nature mutable and extensible, so that appropriate schemata can evolve in the wild, driven by the pragmatics of actual use, rather than by a priori top-down design.

(It should be noted that so-called "Object databases" attempt to address this issue by using object inheritance and other notions borrowed from Object Oriented Programming (OOP) to manage item schemata. However, in most such schemes, an object's schema becomes fixed at the time of its creation, and thus lack the dynamic extensibility that we believe is essential to the present agenda.)

Universal Identity

The third, and perhaps most fundamental requirement of currency of the Universal Database is that each data object have a universally unique identity. It is a fundamental principle of systems design that the first step in ensuring interoperability between two subsystems is the establishment of a common name space between those systems. It is simply not possible to reliably flow information between two venues unless those venues share a common idea of the identity of that information. If we have two namespaces, we have two systems. With one namespace, at least the foundation has been set for a single system. Above all else, the agenda implied by the Universal Database requires a technique for unambiguously preserving the identity of each and every distinct data object wherever in the world its travels may take it.

We have already seen that the relational model does not satisfy this requirement. Fortunately, however, this requirement is neither difficult nor expensive to achieve. To demonstrate this, consider the following thought experiment: Suppose that the administrators of all relational databases in the world go together and decided that from now on, the primary key of all tables would contain a moderately-sized (say, 16 byte) random number. Setting aside the fact that they would all consider such a proposal absurd, let us ask the question of what such a world would be like and how it would differ from current practice.

For one thing, we would run the (vanishingly small) risk that two keys somewhere in the world might be identical. In exchange for this risk, though, we would gain the guaranteed ability to move a row from one table to any other similarly structured table with impunity (we will have to solve the schema problem in some other way). Our database designers will also have to give up the common practice of having primary keys also double as holders of content. That is, we would be deprived of the convenience of making names, or social-security numbers, etc, also serve as the primary key, which in this scheme becomes a pure identifier-any implicit mappings between the key and the identity would have to be made explicit. These would seem to be a small price to pay for solving, once and for all, the identity problem. N.B.: in this hypothetical world, we have not merely provided unique identity to each table, but rather to each and every data item, no matter how modest. Key conflicts in interoperable systems would have, in one sweep, been eliminated as an issue.

Now, of course we are not advocating such a scheme-we can do far better. But our little experiment serves as an existence proof of the feasibility of a general solution to the identity problem, and a suggestion of just how cost-effective such a solution could be.

The U-form - A Universal Data Container

The remainder of this paper outlines in broad strokes the foundations of an architecture for persistent data storage of arbitrarily-distributed information systems. This model, known as the Visage Information Architecture (VIA) is founded on the principle of layered semantics, that is, that lower layers of the architecture should make the fewest possible assumptions about how higher layers will structure or manipulate the data. This principle, which has long been doctrinal in communications network architecture (c.f. the ISO OSI "7-layer" reference architecture), is, we believe, essential to the evolution of a true large-scale information space. Consequently, the basic abstractions defined by this model are extremely simple. Despite (or, rather, because of) this simplicity, we believe that the abstractions defined within the model suffice as the basis of a universal information architecture that can be both rigorously defined and also evolvable.

The basic abstraction of VIA is an abstract data type called the u-form. A u-form is simply a bundle of name-value pairs associated with a universally-unique identifier (UUID). (The name u-form derives from a proposal by Michael Dertouzos that simple name/value pairs, which he termed e-forms, be adopted as the universal messaging substrate among software agents. Our step of adding UUIDs to this scheme in order to support persistent storage yielded the name u-form). A u-form may be conceptualized as follows:

<UUID>:

attribute name 1	value 1
attribute name 2	value 2
...	...
attribute name n	value n

U-forms have the following properties:

• "UUID" is a sequence of bytes that, within acceptable engineering tolerances, is assumed to be unique in the Universe. (How to generate such a value is beyond the
• In addition to the UUID, a u-form comprises a set of attribute name/value pairs.
• Each attribute name/value pair forms a 2-tuple comprising a single attribute name and a single value.
• Each attribute name is a text string of arbitrary length. It must be unique within the u-form.
• Each value is a sequence of bytes of arbitrary length.

The above is the entirety of the definition of the data type "u-form". It is very important to understand that the u-form is an abstract data type, and not a representational format. It is analogous to "integer" or "float," not to "4 byte big-endian integer," or "IEEE Standard 754 32-bit floating point value." Of course, any actual realization of a u-form must be rendered into some particular representational standard. Although the VIA architecture defines such standards at a higher level of semantics, the term "u-form" as it appears in this document always refers to the abstract data type, not to any particular realization.

The purpose of the u-form in our architecture is that of a generic data container or mutable data type. That is, it is the simplest unit of information that can be modified or extended without changing its identity. In this regard, it plays a role similar to that of a variable in traditional programming languages or of an object in object-oriented programming. Unlike these, however, the u-form is a purely declarative data type-it exists entirely in information space. Variables and objects, on the other hand, are not products of information architecture, but rather of system architecture. That is, their fundamental nature is defined in terms of some specific programming environment. An object may be serialized (i.e., described as data) into some external format such as XML. But such a serialization is not itself an object, merely a description of one. Objects as such can only exist in the context of a running computer program. U-forms, on the other hand, are (like integers or XML or relational tables) essentially data. They are message, not medium. Unlike integers or XML, however, each u-form has an intrinsic identity that remains the same even if the contents of the u-form are modified.

It is because of this invariance of identity that u-forms are able to play the role of generic data containers. To understand the importance of this, it is useful to consider an analogy: In the mid 20th century, the transportation of goods was revolutionized by the introduction of a very simple technology: the standardized container. Before containerization, goods traveled "break bulk": Sacks of potatoes were lifted by longshoreman into giant nets and swung by crane into ships' holds. Odd-sized crates were packed one-by-one with more or less skill into boxcars and trucks. With the advent of standardized containers, all of this rapidly changed. The entire transportation infrastructure was redesigned around these containers, with standardized forklifts, and ship holds, and rail cars and trucks. The container literally encapsulated the goods, effectively isolating them from the machinery of transport. This is a very real example of "layered semantics," and one worth examining as we contemplate the Universal Database.

Just as standardized containers made possible vast improvements in the infrastructure and business practices of the shipping industry, just so, the u-form affords the possibility of similar improvements in the infrastructure and business practices of distributed computing. Just as standardization on the byte permitted standard modems, standard integrated circuits, standard disk drive interfaces, etc, etc; a u-form based database architecture permits a great many components to be built in a totally semantically-neutral manner. I can place any data whatsoever "in" a u-form, and change the value of those data at any time without ever creating any ambiguity as to the identity of the container, and without changing in any way the ability of our low-level database infrastructure to manipulate the u-form as such in a uniform manner. It is true that the applications layers will still have to come to some consensus on the interpretation of the data stored within the u-form (just as you and I need to have a shared understanding of how to interpret the text in this document). But by standardizing on the u-form, a great many generic services can be defined and engineered without respect to this higher-level agreement.

Using U-forms

A u-form based persistence model has very different characteristics from traditional database practice, and its practical use requires the discovery and development of many new design patterns. Although we have build numerous practical systems based upon the u-form abstraction, including several fairly large-scale commercial applications, we do not pretend to have made more than a down-payment on the development of these design patterns. However, several such patterns have proven basic, and an understanding of them seems essential to an understanding of the agenda implied by this work. This section provides an overview of several key patterns that have emerged from our work with u-forms.

Relations

The u-form as such is schema-free. Just as the definition of the byte in itself says nothing about how I may use bytes to represent (for example) floating point numbers, so the u-form per se tells us nothing about how to represent higher-level data abstractions. A u-form may have any number of attributes, and their names can be arbitrarily chosen. Although VIA defines standards for many such issues, they are for the most part beyond the scope of this document. However one such notion is indispensable to the practical application of the model, and that is the notion of relations. Simply, a relation is a u-form attribute value consisting of a sequence of UUIDs. The relation is the basic mechanism by which u-forms may refer to other u-forms. The notion a relation is very similar to that of a pointer in standard programming languages. However, rather than depending on the address of the storage location as the means of reference, relations depend on UUIDs, and are thus completely location-independent. One can form relations with u-forms that reside in different databases, or on different computers. Relations can refer to u-forms whose locations are unknown. One can even refer to a u-form that doesn't physically exit.

Those familiar with Lisp and other traditional Artificial Intelligence programming languages will immediately recognize the similarity of VIA relations with the familiar “links-and-nodes” style of programming that is common in knowledge engineering and other branches of AI. Indeed, many of the techniques and patterns developed within this tradition are directly applicable in the present context.

One of the most basic uses of relations is in the creation of u-forms that represent aggregates of other u-forms. Such u-forms, known as collections, are a fundamental unit of organization within VIA. Complex dataspaces are often represented as intricate webs of relations among collections. Since a single u-form may contain more than one relation, such webs are not limited to hierarchical topologies, but rather can form arbitrary (possibly cyclic) graphs. Such graphs can span any number of physical venues, and thus are amenable to the representation of incomplete or distributed knowledge spaces.

Retrieval-time Schemata and Polymorphism

We have already seen that, in contrast to relational tables, a u-form's schema is extrinsic to the u-form. That is, it is not a property of the u-form itself, but rather an interpretation applied to the u-form from the outside. VIA defines a rather detailed mechanism for this, which will not be described here. Suffice it to say that the schema for a u-form is described via a relation to another u-form (known as a role u-form) that provides information about how the u-form was intended to be interpreted. In this way, the schema of each u-form (and thus the precise manner in which it is interpreted) is kept strictly separate (both logically and physically) from the u-form itself.

This fact immediately suggests a very powerful corollary: a u-form may have more than one schema. This is accomplished by relating the u-form to more than one role. Although these roles may not syntactically contradict each other by assigning conflicting meanings to the same attribute name (if they do, the u-form is considered malformed), the various roles that a u-form “plays” may be quite distinct. These roles may complement or elaborate upon each other (such as adding a “annotation” role to a u-form representing a text document); they may be independent of each other; or they may even semantically contradict each other. The ability to add unanticipated roles to existing u-forms is a prime source of extensibility and introspection within the VIA architecture. It becomes possible to write applications that can process data objects that they only partially understand-even to the point of modifying those objects without risk of compromising other applications' continued ability to interpret and modify other attributes of the u-form.

The ultimate effect of this is that schemata can be applied and even defined at retrieval time, rather than when the database is created. The importance of this fact in enabling the creation of evolvable systems is difficult to overstate.

One final implication of this separation of content and form is that it permits polymorphism, that is, the ability to display the same underlying data in completely different ways. In a VIA-based visualization environment, it is quite easy (indeed almost automatic) to create tools that permit one or more users to simultaneously view and manipulate the same data through completely different renderings. For example, one user may be viewing a collection of data rendered on a map, while another user views and edits the same data displayed as a spreadsheet-style table. This capability was developed in a DARPA-funded research project known as Visage Link, and has been applied in a number of government and commercial contexts by MAYA Viz, an affiliate of MAYA Design.

The VIA Repository

The notion of the u-form-being merely a data abstraction-can be used in many ways. The most basic of these, however, is as the basis of a new kind of database technology. We have designed and implemented such a database in the form of a product known as the VIA Repository. Essentially, this is a scalable persistent data store, existing both as an embeddable component and as a network server, whose essential role is the storage and retrieval of u-forms. Although the efficient implementation of such a server is somewhat subtle, its external interface is extremely simple. The basic operations consist of the following:

GET ATTRIBUTE: Given a UUID and an attribute name, the repository returns the previously stored value of that attribute.

SET ATTRIBUTE: Given a UUID, an attribute name and a value, the given attribute is set to the given value.

LIST ATTRIBUTES: Given a UUID, the repository returns a list of attributes that have values in the specified u-form.

There are a few elaborations on these basic operations, mostly having to do with such services as efficient search, retrieving partial values of large attributes, and authentication/security, but the overall model remains extremely simple, both conceptually and in practice.

This repository has been used in a number of contexts and is quite robust both from a stability and a scalability perspective. Detailed performance studies have been performed and are available from the authors.

Replication and the "Grand Repository in the Sky"

One of the most powerful techniques afforded by the use of u-forms - and perhaps the most essential to the realization of the Universal Database - is replication. Replication refers to the ability (through the magic of UUIDs) for the same u-form to exist simultaneously in more than one venue. If two u-forms have the same UUID they by definition are the same u-form. Thus, replication is the act of creating a new instance of a u-form in a different venue. This is to be distinguished from copying, in which the contents of a u-form are moved to another u-form (i.e., one with a different UUID). A replicate differs from a copy in that copies have separate identities, while replicates by definition have the same identity. For example, suppose I move a text file containing the draft of a letter from my office's shared file server onto my laptop and carry it onto an airplane. In doing this, I may have had one of two intentions in mind: Perhaps I planned to edit the draft into its final form. On the other hand, I may have simply wanted a copy of the text to use as the starting point for a completely new letter. In the former case, my intention is for my airborne edits to eventually propagate back to the original version on the file server (and we may have a problem if my assistant edits the same document in my absence). In the second case, however, the copy has assumed a new identity with no continuing tie back to the original. Note, however, that my physical actions are the same in both instances. Only my intentions distinguish the two cases. Most contemporary computing systems provide no structural way to denote this essential distinction. In other words, they do not distinguish copying from replication. This is a serious problem that is well-addressed by the u-form abstraction.

Of course, just because two replicates are defined to be the same does not mean that they are the same. Indeed, it is impossible, even in principle, to guarantee in general that two mutable data objects are identical at any given time. (transaction processing systems can make such a guarantee in a limited way, but only under very constrained circumstances and at considerable expense). Much of our recent research has centered around the problem of how to deal with the inevitability that, in a world as distributed as the one we contemplate, there will always exist inconsistencies in replicated data. The dual recognition that (a) one can use replication to approximate the ideal of having the same data simultaneously accessible in multiple venues simultaneously but that (b) any such approximation will inevitably be imperfect proves to be provocative: How can we make this approximation useful? What exactly is it an approximation of? What compromises will nature force on us, and how can we mitigate them?

In beginning to explore these questions, we begin with a (facetiously named) abstraction called The Grand Repository in the Sky or GRIS for short. GRIS is a persistent store of unbounded capacity that is capable of providing instantaneous, transactionally-consistent read/write data access anywhere in the world. Regrettably, GRIS is physically unrealizable -- it violates various laws of physics. However, by using u-forms and replication, we can indeed build useful approximations of GRIS. Specifically, we envision an unbounded peer-to-peer network consisting of one or more VIA Repositories at each venue. All clients will connect only to local Repositories (thus approximating GRIS's instantaneous access). Repositories will be interconnected by a special kind of agent-specifically, an autonomous data replicator known as a shepherd. The shepherds' job is to implement various business rules concerning the appropriate replication policies to be applied to u-forms whose contents are inconsistent across the venues.

Obviously, the notion of GRIS raises many more issues than it solves. However, one overriding principle has become clear, which we have named the GRIS Principle. It can be stated as follows: In developing clients for the Universal Database, one must never introduce any mechanism that depends upon inconsistent data across replicates of any u-form. That is, although we must, generally speaking, be prepared to deal with inconsistent replicates, we must never depend upon such inconsistency. This deceptively simple dictum has significant consequences. To pick just one simple example, it is common practice for word processing software to store state concerning user preferences such as display options directly in a word-processing file along with the file's contents. In the world described here, however, there is no telling when some shepherd may come along and replicate this information across various users simultaneously viewing or editing that file. On the other hand, if we are always careful to only deploy applications that honor the GRIS Principle, we can rest assured that our shepherds, along with a vast network of other people's shepherds, can safely and independently go about their business of approximating GRIS without fear that doing too good a job may prove harmful.

Dealing with the implications of the GRIS Principle, along with many other such design guidelines have the effect of forcing a much more careful management of the information space-effectively forcing less reliance on system design and a much more rigorous approach to information architecture. Our understanding of such architectures is still rudimentary. Improving this understanding is very likely in the critical path to progress toward the Universal Database.