Data Documentation and Retrieval in a UniVerse® Environment

ABSTRACT

Mergers, acquisitions, and good business practices constantly contribute to the growth of small companies into larger conglomerates. This growth means that software and systems that were sufficient to run a company one day may be insufficient the next. Database systems that were poorly designed or documented during a company's incubation period may become obstacles to additional growth. In this paper, we discuss methods for improving access to poorly documented databases in a UniVerse® environment. We detail approaches involving ODBC access to the UniVerse environment and the usage of UniVerse programs to improve access to the same data using examples from a grocery distributor's current database system.

INTRODUCTION

Even as some companies lose market share and stock value, others remain strong and even grow. This growth means that software and systems that were sufficient to run a company one day may be insufficient the next. Database systems that were poorly designed or documented during a company's incubation period may become obstacles to additional growth. This paper will discuss automated approaches to documenting the tables and columns of a database system in order to improve access and overcome these obstacles. This paper will also cover methods used to access data that is poorly documented.

Motivation

During the growth cycle of most companies exists the inevitable need to implement better software that fulfills new requirements. At the turn of the century, the requirement driving many companies was year 2000 compliancy. The need to modify programs to handle four digit rather than two digit years was an essential part of making it to the new century as a viable company. Given that a major expenditure was to be made on upgrading software, some companies, such as the one discussed in this paper, decided to acquire software that would do more than simply make the company year 2000 compliant.

The software required to run the day-to-day operations of a grocery distributor with products numbering close to 50,000 and ranging from single lipsticks to pallets of soda, and thousands of customers ranging over much of the Midwest, is not at all the type of “install and run” software you might use for handling office documents. Software for running a distribution center is only useful after extensive setup and customization. If any customization or development is to take place after the implementation of this type of software, then the company developers must thoroughly understand the system and the underlying software.

In an ideal situation, a software development company creating a piece of software for sale to one of these companies would document it and understand it before selling it. In the same ideal situation, a software development company that creates and sells a suite of software using a database with hundreds of tables would understand and document the complex interactions of these programs and tables. Each column of every table would be documented and, hopefully with little effort, the developer would be able to tell which programs required each column. Not all programmers are software engineers, however, and not all companies that create software do it using appropriate methods. The type of documentation described above takes time, and time is money, especially in environments where hourly rates charged to customers are in the hundreds of dollars and projects range in the hundreds of thousands of dollars. Given a limited budget, a client company might choose additional functionality over complete documentation and therein lies the root of the problem. When it comes to software systems of any magnitude, incomplete documentation is equivalent to no documentation. Any additional changes to the software or database system must be researched heavily, and the only testing that will suffice is full integration with the "live" suite.

Once a client company has made a decision for functionality over documentation, it is up to the client company's programmers and analysts to take up the challenge of documenting the suite of tables and programs and to integrate them with other business tools. Depending on the size of the database involved, it may be necessary to develop tools for automating documentation of and access to the data stored within it.

Outline of Paper

This paper deals specifically with the documentation and access efforts of a grocery distributor. In order to describe the obstacles they faced and the methods they used to overcome them, it is first necessary to describe the database product they use. Once the details of their situation are clear, we will review the problems they faced in accessing the information in their database from outside the programs they purchased. We will examine why standard methods existing for accessing their data worked poorly and study methods they used in their stead. The final part of this paper discusses the results of the distributor's efforts and their plans for further development. Although this paper presents a specific documentation scenario, the insights and approaches presented are applicable to other environments.

Throughout this paper we will examine:

1) The methods for data documentation and access to information stored in a UniVerse® database.
2) The shortcomings of using UniVerse ODBC connectivity as a method for accessing a multi-valued database such as UniVerse.

3) Methods that improve ODBC client access to UniVerse by using documentation to compensate for naming and ODBC access challenges.
4) The use of a semi-automated documentation and access system called Unity on the UniVerse database.
5) The development of UniVerse source extraction tools that
    a. Improve ODBC access to UniVerse.
    b. Dramatically increase the efficiency in constructing reports and data access programs on the host system.

The contribution of this work is an evaluation of the shortcomings of standard access technologies such as ODBC, and a description of methods for annotating and documenting data that improve programmer efficiency when manipulating and extracting data from complex systems.

BACKGROUND

Business environment

The software environment used by the grocery distributor, which we will call GD for short, is a database system under UniVerse comprising over 500 data tables with a total size approximating 80 gigabytes. Each of these data tables has anywhere from 0 to over 200 data columns. The number of virtual data columns in each table is not limited in any significant way. The particular business software being used, like most others, evolves on a day-to-day basis as a result of shifting priorities and business goals. Much of the development takes place on the UniVerse system and is done by company programmers, so any improvements to data documentation should be available to users and developers using UniVerse as well as those using Open DataBase Connectivity protocol (ODBC) to connect to the database.

UniVerse Databases

UniVerse is a relational database environment developed by IBM [1] with built-in ODBC connectivity and its own programming language called UniBASIC. Each database under UniVerse is composed of a number of tables. UniVerse tables are not defined using schemas; therefore they are not constrained to hold a given number of columns at the time they are created. UniVerse separates the data portion of the table from the access portion of the table. In order to link the two aspects of the table, each table has a dedicated table dictionary that defines the columns available for the access layer. In comparison to a relational database system, the dictionary can be seen as a "view" into the table. While relational views can only be defined using SQL statements, dictionary items provide methods for performing joins and other database algebra without using SQL. An entry in the table dictionary contains the information required to retrieve and display the data from the column it pertains to in the table. This information can include formatting instructions, conditional statements, and a label for the data, among other things. Each dictionary item describes a column of the table. Each column available for reporting is a virtual column because of the separation between the access layer and the data layer. These columns can be calculated based on data internal to the table, calculated independently of the data in the table, or based entirely on data from another table. The table dictionary does not always contain an entry for each column in the table but may contain more than one entry for each of the columns it does reference.

There are several different types of dictionary entries. Table 1 provides a sampling of dictionary items for the dictionary of the customer table used by the grocery distributor.

Dictionary Record ID	col 1	col 2	col 3	col 4	col 5	col 6	col 7	col 8	col 9	col 10
CM.NAME30	S	3	NAME		S				T	20
CSTM.NAME.ADDR	I	TRIM(PNET.INFO&lt;1,2&gt;:' ':PNET.INFO&lt;1,1&gt;)		PNET CONTACT	25L	S				
CUST.NAME	A	0	NAME		S			A;IF 100 # "" THEN 100 ELSE N(NAME)	R	20
EU.CUS.NAME	S	3	CUSTOMER NAME		S				T	30
FNAME	D	8		NAME XREF	6L	M	M7			
FULLNAME	S	2	BILLTO NAME		S			A;01:"*":02 TCM;C;3;3	L	20

Table 1: A sample listing of dictionary records from the customer file.

The columns labeled "col 1" through "col 10" represent attributes of the UniVerse dictionary items. Attributes in UniVerse correspond to columns of a data table. The type of the dictionary entry is determined by the first attribute of the entry (col 1). This type determines the structure of the rest of the entry. Appendix 1 gives a series of structures for the most common dictionary entry types in UniVerse. Column 2 for "S" type dictionaries determines which data column is referenced by the dictionary entry. Notice that `CM.NAME30` and `EU.CUS.NAME` reference the same column. The other columns of the dictionary entry determine the difference between the resulting data when the dictionaries are used.

The native programming language of UniVerse is UniBASIC, a compiled version of BASIC with extensions used to work with the databases directly. Programmers do not need to know the details of the hashing algorithm used to access the data portion of the table, but they do need to know in what column number the particular piece of information they are working with is stored. Generally, dictionary items are not used within UniBASIC programs. Along with UniBASIC, UniVerse also utilizes several procedural languages for automating data flow and user interfaces. UniVerse also provides a powerful query language for generating reports. This query language is called RETRIEVE [2]. Programmers can design and execute RETRIEVE reports within UniBASIC by painstakingly concatenating strings that define a RETRIEVE statement using dictionary items. There is no built-in utility in UniVerse to assist in this process. Along with RETRIEVE, an SQL variant is also available within UniVerse for executing local queries. This SQL variant is used by the ODBC server to fulfill client requests. All of these tools are primarily used through telnet clients, which means there is no graphical interface for them. Even so, these tools make the UniVerse environment a powerful tool for building enterprise software.

UniVerse presents several different methods of accessing the information in a table. A simplified record from the customer file of the grocery distributor might look like this:

```
    01~005267~JOSE JIMENEZ~IA~~O~01~JJ|52241|ST*IA
```

The delimiting characters in a UniVerse record are nonprintable ASCII characters so they have been changed. The end of attribute marker has been replaced with a "~" and the end of value marker has been replaced with a "|". If the given record had the key of "`01*005267`" and the name of the table was "`CM`," then the data could be displayed by entering a RETRIEVE command that uses two of the example dictionary entries listed previously. A simplified form of the statement is

```
    LIST <filename> [<id list>] [<fieldname>]
```

An example command would be:

```
    LIST CM 01*005267 CM.NAME.30 EU.CUS.NAME
```

Executing this command would result in the following output:

```
CM........ Name......................... CUSTOMER NAME.....................
 01*005267 JOSE JIMENEZ                  JOSE JIMENEZ
```

Note that while the information printed in each of the two name columns is the same, the length of the data presented is different and so is the column header. The details of that transformation are in the dictionary entries. Accessing the same information via UniVerse BASIC would be done with the following code:

```
OPEN "CM" TO CM ELSE STOP
READ CMR FROM CM, "01*005267" ELSE STOP
PRINT "CM........  ":
PRINT "Name......................... ":
PRINT "CUSTOMER NAME....................."
PRINT "01*005267" 'R#10':" ":
PRINT CMR<3> 'T#20':
PRINT CMR<3> 'T#30'
STOP
```

UniVerse BASIC uses the convention of using angle brackets, < and >, to designate accessing information within a variable that has multiple attributes and values. `CMR<3>` references the third attribute in the variable `CMR`, which is a record read from the CM file using record id "`01*005267`." The code then prints out a header and the data similar to the output of the RETRIEVE statement. Using SQL, the same could be accomplished by the statement:

```
SELECT CM.NAME30, EU.CUS.NAME FROM CM WHERE KEY = '01*005267';
```

Unlike relational databases, UniVerse has native support for accessing multi-valued information. In order to do this using RETRIEVE, it is only necessary to add the name of a dictionary entry that references a multi-valued attribute such as FNAME. Listing that value in the same file would return:

```
CM........ Name..

 01*005267 JJ
           52241
           ST*IA

1 records listed.
```

Since the attribute is multi-valued, RETRIEVE displays the information in each value of the attribute on a separate line. Accessing the same information using UniVerse BASIC is similar to the method for accessing a particular attribute, only the angle brackets contains a further parameter to indicate what value of a given attribute is desired. For example, `CMR<8,2>` would reference the data "`52241`" because it is the second value of the eighth attribute.

In a UniVerse environment, there is a special entry in the dictionary for each table that determines what dictionary entries, and therefore what data, are accessible using ODBC. This dictionary item is named "@select." UniVerse SQL relies on the @select entry to describe what columns are accessible, the same way that the ODBC service does. This entry in the file dictionary is generally updated manually. The @select entry of the CM dictionary might look like:

```
P~KEY EU.CUS.NAME FNAME CSTM.NAME.ADDRESS~UPDATED BY JMJ ON 3/3/03
```

An @select record such as this would indicate to the server that the columns `KEY`, `EU.CUS.NAME`, `FNAME`, and `CSTM.NAME.ADDRESS` were available for access via ODBC. The SQL statement "`SELECT * from CM where KEY = '01*005267';`" would result in only the columns listed in the @select entry being returned. The column FULLNAME, while in the dictionary and available for RETRIEVE statements, would not be available to ODBC clients or SQL statements. Figure 1 displays the architecture of this environment.

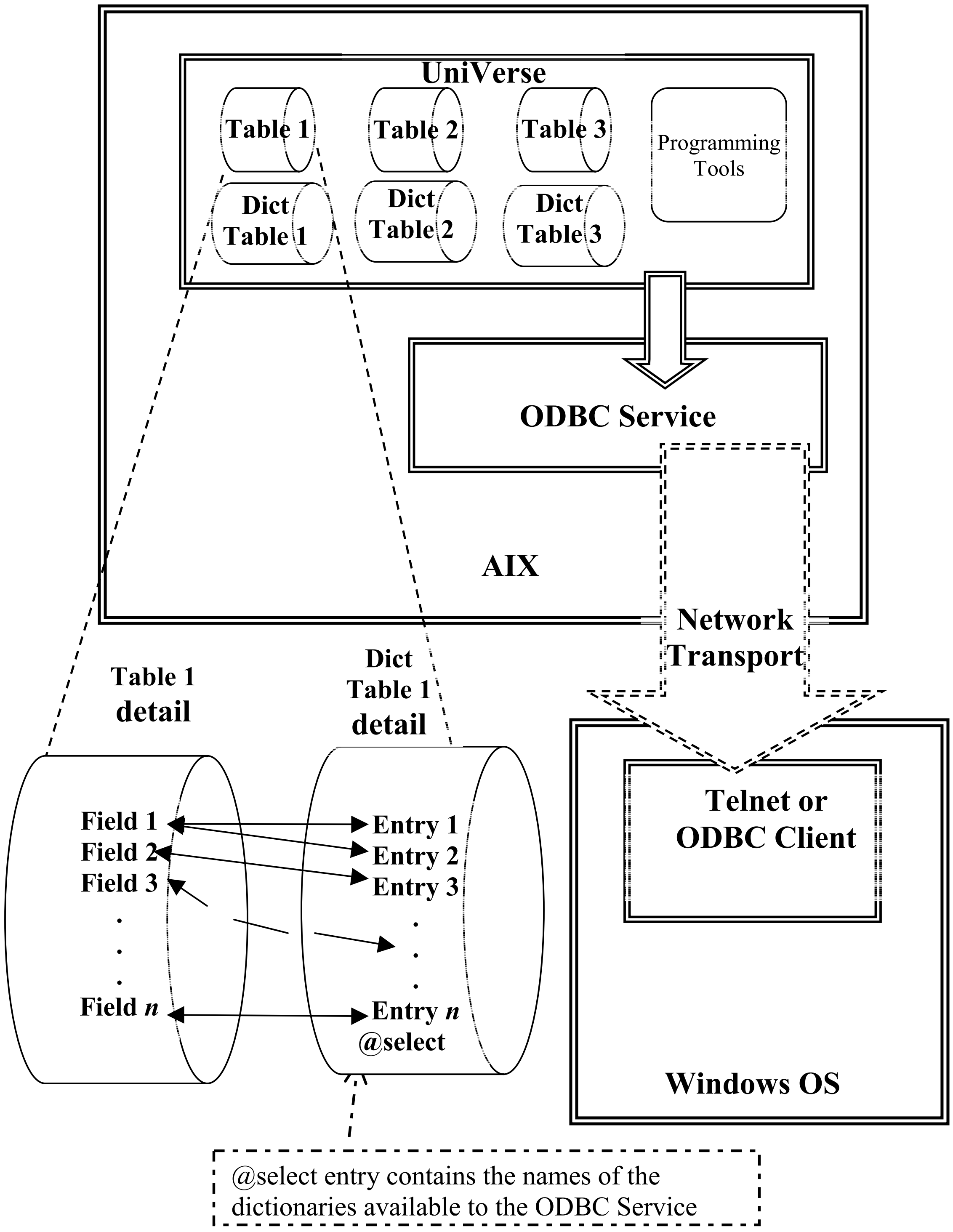


Figure 1: The UniVerse to ODBC connection. Note that Field 1 has two dictionary items: Entry 1 and Entry 2, which reference it to produce a value.

THE PROBLEM OF INADEQUATE DOCUMENTATION

At GD, as at many other companies, it is necessary to examine information to assess buying patterns, employee performance, and other factors that determine success or failure. Aside from projects that are required by changes in operational methods, reporting is one of the major consumers of development time. Some standard tools, such as Crystal Reports®, could ideally be used in preparation of reports. In order to use a tool such as Crystal Reports, it is necessary to setup ODBC connectivity or use other methods to provide the data from UniVerse to that tool.

The name of each column visible to an ODBC client is the name of the column's entry within the dictionary for the table. Within UniVerse, this name is generally only visible to programmers using it within a procedural language to generate reports, and the name is not used for programming in UniVerse BASIC. The actual label for a column on a report generated with RETRIEVE or one of the other reporting tools available within UniVerse is the column header stored within the dictionary record. Due to the separation between the name of the item and the displayed column header, names for the dictionary items tend to be arcane abbreviations at best and numbers at worst. For example, while well-named items such as "`NET.COST`" appear in the dictionary of the products file in the grocery distributor's database, items named poorly, such as "`CW`" or simply "`2`," also appear in the same dictionary. Some of these names follow a well-established naming convention of naming the column after the number of the column in the physical data file, but to an end user this name is useless. Other naming conventions might use an abbreviation that includes only the first letters of the text that describes the column contents. Again, this type of naming convention is useless without proper documentation to back it up. In order to improve an ODBC client experience, more information than is present within the name of a column is required. If the database comprised only a few tables and a few columns, it would be a simple matter of changing the dictionary entries or creating new ones with significant names. In a database of this size, however, with over 500 tables and hundreds of dictionary items for each table, changing names is not a task undertaken lightly.

Possible Solutions

The task of providing quick and easy access to information stored in GD's database system eventually fell to the developers and analysts at GD. For a period of time, GD relied on their software vendor to integrate the new database. It soon became apparent, however, that this was too costly in time as well as money. With delays ranging into months before the vendor would look into a project request, GD soon began to develop in-house expertise in order to fulfill user requests for information. Even with this approach, some projects were still too costly to consider.

UniVerse's File Extraction Tool

One of the ways investigated to get information from UniVerse to other environments included extracting information from the data tables into text files in order to use other data-analysis tools. UniVerse has a built-in tool called `UVEXPORT` [3] that generates files in either comma-separated ASCII or Lotus Notes format by using the dictionary entries of a table to define columns of the exported file. There are many limitations in `UVEXPORT`. For example, `UVEXPORT` does not allow for searching for a dictionary item. Some dictionaries within the test environment hold hundreds of column definitions. With `UVEXPORT`, searching for a particular dictionary entry is a matter of searching through an unfiltered list of every record in the dictionary. Poor naming practices often result in problems identifying the appropriate column for an extract. As stated before, a name such as “`4`” means very little to a user. The column header information is not displayed by `UVEXPORT`, so this additional information is not available to a user when defining an extract. Additionally, extracting multi-valued data causes the program to generate a file with variable record layouts that is difficult to read and parse. A given datum might fall within the third column on every line unless the second datum was multi-valued. In that situation, it might fall in the third column if the second datum had only one value or the fourth column if the second had two values, and so on.

UniVerse 10

A more efficient solution is to extract the information into XML, eXtensible Markup Language, which would handle the multi-valued aspect of UniVerse better. UniVerse version 10 provides the functionality required to generate XML with upgrades to its RETRIEVE language allowing the use of standard transformation engines to access the information. The problems of poor naming practices still come into play, but the XML generation relies on the dictionary items for related multi-valued columns being set up properly. In a situation where these dictionary items were not set up with this use in mind, the XML retrieval of the information ends up creating incorrect groupings of columns.

USING ODBC TO ACCESS AND DOCUMENT DATA

Due to the shortcomings of the standard tools when it comes to facilitating reporting, it was important to improve documentation and access using other methods. Using an ODBC client connection one might be able to annotate the objects of the database with natural language descriptions such as those proposed for the Semantic Web [4]. However, several obstacles presented themselves when we attempted to use an ODBC client to assist in database documentation. The first problem was that very few columns were presented to ODBC clients from UniVerse. The @select entry for the tables had very few entries. These entries had been selected for specific reports by the software vendor. In researching methods to increase visibility of columns to ODBC clients, one approach we considered was to simply add every dictionary entry for a given table to its @select dictionary entry. This “brute force” method was tried, but resulted in very little success. Tracking down the problem was not difficult. Columns described in the dictionary as right justified are interpreted by the ODBC service as numeric. When these columns contain information other than numbers—that is any alphabetic characters—they cause a failure on the client because of the data-type mismatch. This mismatch causes any queries on the table to fail completely and not return any data. Since this is not a configurable setting on the ODBC driver for UniVerse and there is only one driver for UniVerse available, it was necessary to solve this problem before continuing.

Improving Access for ODBC Clients

The first step toward solving this problem was to write a UniVerse program to populate the @select dictionary entry with fields that would not cause a failure such as already described. In order to do this, several things had to be considered. Right-justified mixed-type columns were not added to the @select entry. Dictionary entries that referred to non-existent programs or other dictionary items that were themselves invalid were also omitted from the @select entry. Using this program, the @select entries for the database were populated with valid dictionary items. While some dictionary items were omitted, this omission was acceptable because the dictionary generally contained more than one entry per actual column of the table.

Documentation for ODBC Clients

The second problem holding up documentation and making new development and modification of existing programs difficult was identifying what each available column contained. With names such as "`CW`" or "`2`", some columns were difficult to classify. This is where the ODBC client has much less information available than does the UniVerse dictionary itself. When creating a dictionary entry, a user can enter a descriptive field name. This descriptive field name is used as a report column header by built-in reporting tools such as RETRIEVE. This text, because of its use, generally has information encoded in it that is not in the name of the entry. For example, the "`CW`" entry has a column header of "`Catch Weight`" which, in industry lingo, refers to an item that is priced by the pound, such as a bulk cheese or frozen meat. This header text gives the additional information needed to decipher what data is stored in the column. This information is not available via the UniVerse ODBC interface even when requesting metadata information for the column.

The column header text of a dictionary entry, along with the name of the dictionary entry, can be used in assigning a *semantic name* to the column. We define a semantic name as an annotated name associated with a database object that provides more descriptive meaning to the end user. One method to present this semantic name to an ODBC client would be to create new dictionary entries for each column incorporating both the original name and the header text into the new name, using the character "`@`" as a separator. The "`CW`" field would then be presented as "`CW@Catch Weight`." The "`@`" character was chosen as a delimiter because it is a valid ODBC character. Other valid characters could serve the same purpose. This methodology was incorporated into the UniVerse program that populated the @select entry with viable dictionary entries. ODBC clients accessing the database were then presented with an increased number of well-named columns.

Although this method increases the number of columns with meaningful names accessible by ODBC clients, it has the undesirable affect of changing the dictionaries in the UniVerse environment. The dictionary entries created with the new semantic names are visible to users in the host environment, who now see two dictionary mappings to the same data: one with the original poor name, and the new one with the better semantic name. The duplicate entries confuse documentation and usage at the host-level. Further, only documenting columns that are accessible using ODBC does not help in situations where host-level data is not accessible via ODBC. Thus, documentation techniques that do not modify host-level data and dictionaries are desirable.

Documentation Using Unity

One non-obtrusive method for database documentation is using Unity [5]. Unity starts with a global dictionary that holds information about entities or objects it is likely to encounter within a given database. The global dictionary relates elements of the database to English words. Unity then uses ODBC to access a data source and retrieve its schema information. Using standard ODBC interface calls, Unity explores the tables of a database and creates an internal representation of its structure in a XML document called an X-Spec. Unity semi-automatically assigns semantic names to database objects using the global dictionary. Assigning a semantic name improves the documentation for a given column and allows for relating columns from multiple tables and databases using semantic names. The documentation process of Unity is similar to the annotation applied to the human genome. In this case, data source information extracted through ODBC is semi-automatically annotated to improve documentation and readability. The major form of annotation are semantic names which provide the user with the ability to access data using more meaningful conceptual names that are then mapped to the names and structures used in the data source. An X-Spec is an XML encoding of the database schema and semantic annotation that allows for systematic documentation, querying, and integration of data sources. Figure 2 shows a partial DTD for an X-Spec.

```
<!ELEMENT XSPEC (System_Name,...,TABLE*)>

<!ELEMENT System_Name (#PCDATA)>
...
<!ELEMENT TABLE (System_Name,Semantic_Name...,FIELD*,KEY*,JOIN)>
<!ELEMENT System_Name (#PCDATA)>
<!ELEMENT Semantic_Name (#PCDATA)>

...
<!ELEMENT FIELD
      (Semantic_Name,System_Name,Field_Type,Field_Size,...,
      Comment,...,Function_String)>
<!ELEMENT Semantic_Name (#PCDATA)>
<!ELEMENT System_Name (#PCDATA)>
<!ELEMENT Field_Type (#PCDATA)>
<!ELEMENT Field_Size (#PCDATA)>
<!ELEMENT Precision (#PCDATA)>
...
<!ELEMENT Comment (#PCDATA)>
...
<!ELEMENT Function_String (#PCDATA)>
<!ELEMENT KEY (Key_Name,Key_Type,Key_Scope,Scope_Name,FIELDS)>
<!ELEMENT Key_Name (#PCDATA)>
```

```
<!ELEMENT Key_Type (#PCDATA)>
<!ELEMENT Key_Scope (#PCDATA)>
<!ELEMENT Scope_Name (#PCDATA)>
<!ELEMENT FIELDS (FIELD_NAME)>
<!ELEMENT FIELD_NAME (#PCDATA)>
<!ELEMENT JOIN (Join_Name,...)>
<!ELEMENT Join_Name (#PCDATA)>
...
```

Figure 2: A partial DTD for an X-Spec.

Documenting a host database using Unity is non-obtrusive because all schema information is extracted using ODBC. Documentation takes the form of annotation using semantic names. Since the annotation is stored in separate X-Spec documents from the host, the host schema and dictionaries are never modified. Extraction queries can be generated through Unity's graphical user interface using semantic names that are then mapped to the system dictionary names when the query is actually executed. Using Unity as an extraction and report generation client improves programmer efficiency as semantic names are easier to manipulate than dictionary item names.

Similar to other ODBC clients, Unity is unable to get the complete picture of the UniVerse schema information available in dictionary entries. Recall, that dictionary entries have a column header that is used for reporting in UniVerse, and the column header has a more meaningful name than the abbreviated name of the dictionary entry available to ODBC clients. Unity's semantic annotation process yields very poor results when the dictionary names are such arcane abbreviations. By providing access to the column headers as well, the annotation process yields much better results. Since the column headers are not retrievable using ODBC calls, workaround methods are required.

The first workaround discussed previously is to introduce new dictionary entries at the host-level with names that are a concatenation of the dictionary entry name and the column header. However, this approach is undesirable because of the modifications to the host system. A second approach is to write UniBASIC programs to use the information available from within UniVerse to build the X-Specs and retrieve the additional data from the dictionary tables themselves instead of going through ODBC. Then, any information that could not be extracted through ODBC can be placed in the `comment` field of the X-Spec and is accessible when performing annotation. Once the X-Spec is generated, database information can be disseminated more readily.

A UniVerse BASIC program was developed to create X-Specs that contained all of the information necessary and available for documenting a UniVerse database. Unity was then used to import the data from the X-Spec generated by the UniVerse program, and it allowed for documenting the database without requiring ODBC connectivity. Using the X-Specs with the annotated column names, Unity presents an ODBC client with improved access to the UniVerse data using standard ODBC calls.

Using a database annotation tool such as Unity is valuable because it semi-automates the documentation process. Better documented databases and more meaningful schema constructs, result in more rapid development of reports and data extractions. However, like all ODBC clients, Unity suffers from the inability to access all schema information available in UniVerse dictionaries. Host programs that build an X-Spec are useful for extracting and packaging this schema information so that it can be manipulated by other systems.

The major weakness in building X-Specs at the host-level and exporting them to other systems as documentation is that the documentation generated is only useful outside of the host system itself. In organizations like GD, there is need for documentation to aid external report generation using ODBC clients, and for documentation that aids development with the UniVerse environment itself. The following section discusses how leveraging meaningful names can be used to build a better extraction tool than `UVEXPORT` for use in UniVerse itself.

HOST-LEVEL DOCUMENTATION AND EXTRACTION

One of the goals of the documentation process is to increase documentation and access available to not only ODBC clients but to users within the UniVerse environment as well. One of the most important lessons learned from attempting to develop documentation for the UniVerse database was that more information needed to be presented to a user or programmer in order to decide whether or not a given dictionary entry described the desired column. Given just the column header in addition to the dictionary name, a user could more easily describe a required data extraction far more easily than with `UVEXPORT`, the built in UniVerse tool. A better tool would be able to:

1) Create files in comma- or tab-delimited format.
2) Generate XML documents.
3) Select specific rows from a source table.
4) Allow users to build export definitions easily.
5) Allow advanced users to define virtual columns.
6) Handle multi-valued data.
7) Emulate SQL's Group-By feature.
8) Leave room for future development.

In designing the described tool, flexibility is important. Flexibility is achieved by splitting the tool into two logical parts. The first part is the user interface. This interface allows the definition of the relationship between a source file and an export file. The second part is the workhorse of the tool. It interprets the details of the relationship between a source file and an export file described by the user and generates the appropriate export file.

Selecting the Information to Be Extracted

Some of the critical portions of the interface tool involve the method for selecting specific rows from the source table and the method of selecting columns to be exported. UniVerse allows for selecting a specific set of records to work with using a statement that is similar in structure to an SQL statement. The basic difference in the two types of statements is that an SQL select statement actually retrieves the data as well as describing what data to retrieve, whereas a RETRIEVE select statement acts as the `WHERE` portion of the statement and prepares the data to be processed by a RETRIEVE `SORT` or `LIST` statement. This select statement is of the form:

```
SELECT <<file_name>> [[WITH <<column_name>> <<comparison_metod>>
[[<<value>>]].
```

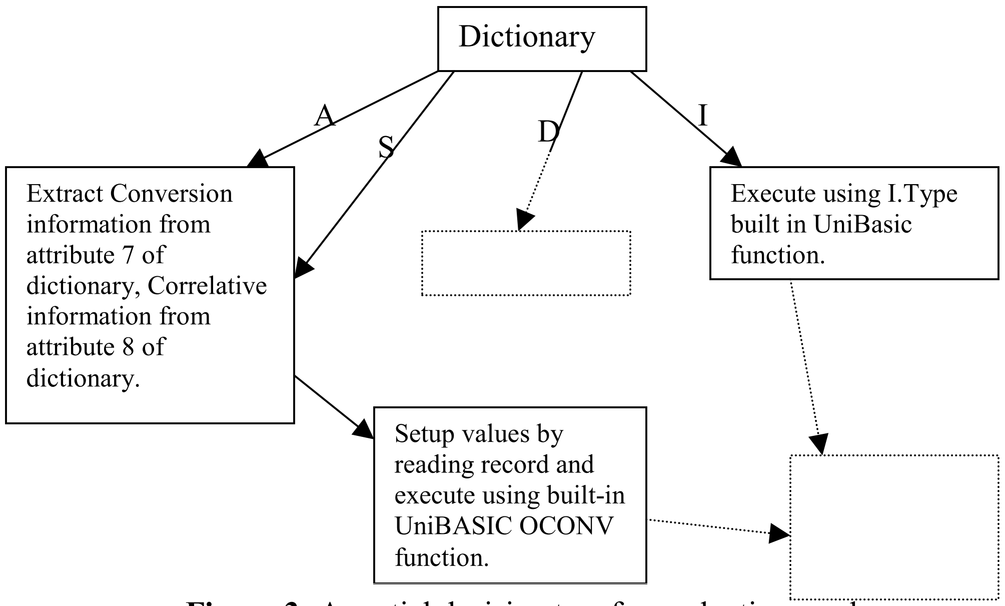
For example, selecting customers who live in Iowa from the Customer Master file called `CM` could be done by entering the RETRIEVE select statement `SELECT CM WITH STATE EQ "IA".` This, of course, only works if the column the state is stored in is called "`STATE`". The "`WITH`" clause is essentially the same as the SQL `WHERE` clause. The "`EQ`" stands for equals and can be replaced with the standard equal sign "`=`". When prompted for a select statement, the user has the option of entering a statement to be executed at UniVerse's main shell that will result in a set of record keys for the source table. The user, if familiar enough with the environment, can simply enter one or use the interactive features of the tool to build one. This feature prompts the user for each of the elements of the select statement and makes it easier for a user with limited or no understanding of the language used by UniVerse to create a select statement.

Besides isolating the user from constructing statements in RETRIEVE syntax, the major advantage of the tool is the user builds the query using semantic names instead of the names of dictionary entries. Using semantic names makes filtering and searching for the appropriate columns much quicker and easier, and drastically reduces query generation time. In contrast, a user who wants to generate the same query using a regular ODBC client (with no source annotation capabilities) would have to look through an unfiltered list of columns with names that may or may not be useful. This list of columns may not contain the column required if it is not in the @select entry, or at best, the user will have to search through a list of poorly named column names to determine the appropriate column to select. In this case, the column named `STATE` is hard to miss. However, if, as in many cases, the relevant columns are badly named, then the ODBC client would not be able to present enough information for the user to select the appropriate column. On the other hand, the UniVerse program described allows the user to filter the list of columns by entering a text search string, which is used to filter the columns both by name and by column header. Both of these pieces of information are displayed to the user when selecting a column to use for selection or display purposes. The UniVerse program also allows the user to quickly see a sample of the values in a particular column when building a select statement interactively. These features make the UniVerse program significantly more usable than an ODBC client.

Advanced users entering a select statement can create a program or use other functionality within UniVerse to create a list and enter any string that can be executed at a UniVerse prompt in place of this selection statement. The interface program relies on the user to validate the output of the statement. So long as the string generates a list of records to be used for the export, the string can do any additional work required by the user. One example would be prompting for input of variables such as a begin and end date to use for reporting. An advanced programmer can use UniVerse BASIC to write a program that generates a select statement that takes into account the day the extraction job is being executed, counts back to the previous Monday, and gets a week's worth of records from a particular file. This program would be useful in selecting the information about the previous week's sales dollars from a detailed file.

The host extraction tool never modifies any dictionary, so its use is non-obtrusive. Further, the user has the option of creating a virtual column by specifying the details that would normally be stored within the dictionary of the table. This virtual column is never written to the table's dictionary and can only be changed from within the interface program, which limits the possibility that a dictionary being changed will affect a stored extract job.

In order to determine the value for a given column, the corresponding extraction program has to build on some of the strengths of UniVerse. It uses a straight-forward algorithm that accomplishes much of what UniVerse's RETRIEVE tool does. Figure 3 shows a partial decision tree for the algorithm. See Table 1 for examples of dictionary records.



Dictionary
A
S
D
I
Extract Conversion information from attribute 7 of dictionary, Correlative information from attribute 8 of dictionary.
Execute using I.Type built in UniBasic function.
Setup values by reading record and execute using built-in UniBASIC OCONV function.

Figure 3: A partial decision tree for evaluating a column

The decision tree in Table 1 does not show some of the steps required to evaluate columns that reference other columns. In order to evaluate these columns, it is necessary first to de-reference any named columns into their respective values and then continue to evaluate the desired column using these values.

The UniVerse programs created for this project directly access UniVerse data that may not be available via ODBC. Similar to SQL views, an extract job or group defines a view into the database that allows the definition of virtual fields, which are not in danger of being changed outside of the extract interface. The programs take advantage of information not accessible to ODBC clients or other programs outside of UniVerse, such as the column header of the dictionary. This information is used as in-place documentation of the data contained within the table being accessed. Developers or users can easily change the content of an extract job without having to sift through code or modify any programs. The ease with which extract jobs can be changed decreases the amount of time it takes to get information out of the database.

RESULTS

Before executing the dictionary verification program to update dictionary items, one of the files used to test ODBC connectivity had 57 columns available for queries via ODBC and SQL. Once the program was executed selecting the update option, 521 columns were available for queries. The additional columns represented dictionary items accepted as valid for ODBC and SQL access. A simple increase in the number of columns available is not necessarily an improvement if there is no improvement in the quality of the information available. The benefit of the additional columns and the improved naming methodology became apparent when accessing the fields using Unity. Determining column meaning was much easier with the new data. Quantifying this improvement was not attempted; however, comparing a label such as "`4`" on a dictionary item to a label of "`4@TAXCODE`" on the column suggests dramatic improvement. The first label may hint at the column number from which the data came. The second label indicates the data in the column corresponds to tax codes. Hence the second label is much easier to work with and understand. Unity's semi-automated naming algorithm can use the more-verbose column name to better assign a meaningful semantic name for the column. Thus, improved documentation in the form of better names makes it easier for UniVerse and ODBC users to query the data and perform advanced manipulation of the data such as required for schema mappings [6].

The new dictionary entries increased access for the ODBC clients but still suffered from having arcane names and had the undesirable affect of modifying the host dictionaries. A better solution building on the strengths of Unity and UniVerse came from the creation of X-Specs using the UniVerse program. In X-Spec creation mode, the UniVerse program did not create new dictionary entries, but instead exported the information not available through ODBC using the X-Spec. The X-Spec can then be manipulated by clients to perform documentation, annotation, schema matching, or query generation with information that would not be accessible using ODBC. X-Specs allow sharing of schema information without providing access to the data. X-Specs contain database annotation that is easier to understand, share with management, and use in schema mapping and evaluation tasks.

The documentation of the database performed with Unity lead to the conclusion that the text found in the column header of the dictionary entries for each file was, in effect, in-place documentation that could be exploited to increase productivity for programming requests at the host-level as well. The availability of this information led to the development of the extraction programs that have drastically decreased development time for new data extractions. In general, the programs allow a developer or a savvy user to create a data extraction routine that can be stored and repeated as needed. With the new tool, simple extractions can be designed and completed by users. The extraction can then remain static or be modified quickly and easily by a user or developer without the need to modify or write a single line of code. Previously, each data extraction was requested separately from a report request and was completely static when coded by a developer. Paper reports were the norm for new development; useful information was trapped on paper, where it could not be manipulated easily. With the new tool, using simple FTP protocols, the generated file can be copied to a client system and then manipulated with standard tools such as Excel, Lotus Notes, or any other product that can import ASCII or XML files.

GD was recently acquired by a larger, nationwide, distributor. One of the initial requirements of integration was the need for additional reporting. One of the main benefits of using the new tool was the ability to focus on the details of the individual data columns required in each report, as opposed to the formatting of the report or the data extraction methods themselves. Using the column headers as in-place annotation, the developers at GD quickly identified the appropriate columns and presented new reports for final approval in a matter of hours, rather than the weeks it would have normally taken. For example, consider the simple task of categorizing sales by a new set of customer categories. In order to do so, it was necessary to add a column to the Customer Master file to hold the new category information and to create a report to sort out the information. Using RETRIEVE to generate the report necessitates that the person creating the report know or be able to find the correct column names to use. The new tool was used to create a file with the necessary information in it. The new tool simplified the selection of the columns and the data to be reported. The extraction routine that was designed is now run weekly. The generated file is then imported into Excel and emailed, mostly untouched, to a number of users. The development time for this project was limited to that required to add the column to the customer table. The report design took minutes rather than the hours or days it would have taken before.

FURTHER DEVELOPMENT

The X-Specs generated by the UniVerse programs provide information about the database not previously accessible via ODBC or SQL. However, there is more information that can be displayed and presented. For example, better support for multi-valued columns is useful. Some of this information can be culled from the dictionaries and other standard documentation. One of the major benefits of using Unity to document the information available in an ODBC or otherwise accessible data source is Unity's use of semantic names. The semantic name, once defined, is a user's first point of contact with a given column. If enough information is encoded in the name, then the user does not need to look at the comments of a column to determine what it contains. To help improve the automated assignment of semantic names, Unity may be modified to access the information required wherever it is stored in the data source.

In the next year, in order to further integrate into the new company it is now a part of, GD will be moving to a new software system. This transition will be completed more efficiently with the new documentation and access tools provided. The new system uses indexed, sequential access files. Each program that accesses the information in the files has to be written from scratch. Yet, there may be enough information in some source code files to write a generalized tool that allows a user to generate reports without resorting to creating new programs. Accomplishing this using the tools provided in the new environment is a future goal of this project.

CONCLUSION

The goal of this project was to determine better ways for documenting and accessing data in a multi-valued database such as UniVerse in order to make report generation and data extraction more efficient. The first result was that standard ODBC access is often insufficient because not all columns are presented to ODBC clients, the names presented are poorly constructed, and insufficient schema information and comments are available through ODBC calls to determine the meaning of the data.

Using Unity to semi-automatically document the databases was an improvement, but without access to the additional schema information available in UniVerse dictionaries, the annotation process was largely unsuccessful. A simple method to improve ODBC access involves adding new dictionary items with more meaningful names for use by ODBC clients. However, this has the undesirable affect of modifying the host system and making the dictionaries even more complex.

The solution was to develop tools and algorithms that allowed for efficient documentation and extraction without source modifications. Building X-Specs as XML encodings of schemas is valuable as it allows schema information to be provided to external systems. X-Specs allow for better annotation and schema matching as they provide additional schema information not accessible via ODBC. We also described an improved extraction tool for the UniVerse environment that used the idea of presenting more documentation to the user to greatly improve report generation efficiency. These tools and techniques are applicable to other UniVerse databases, and are increasingly important as data is integrated across an enterprise.

Time is money, and now that GD is a part of a bigger company, even more data integration projects require them to have a good idea of what information is stored in their database. Using X-Specs and other methods of documenting the database in conjunction with sophisticated semantic naming rules will help to continue their fast-paced development cycle.

REFERENCES

[1] IBM. UniVerse Overview. http://www-3.ibm.com/software/data/u2/universe/

[2] IBM .UniVerse Guide to RETRIEVE. http://www-3.ibm.com/software/data/u2/pubs/library/952univ/70-9005-952.pdf

[3] VMARK. VMARK Technical Bulletin Part No. 74-0074 UVEXPORT: The UniVerse Export Facility. VMARK Software Inc., 1993.

[4] B. Katz, J. Lin, and D. Quan. Natural Language Annotations for the Semantic Web. *Lecture Notes in Computer Science 2519*, pg. 1317, November 5, 2002.

[5] R. Lawrence and K. Barker. Using Unity to Semi-Automatically Integrate Relational Schema. Demonstration at *ICDE 2002 - 18th International Conference on Data Engineering*, pages 329-330, 2002.

[6] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, Volume 10, p. 334-350, 2001.

[7] VMARK. UniVerse System Description Part No. 70-9003-931. VMARK Software Inc., 1996.

APPENDIX 1:
UNIVERSE DICTIONARY ENTRY TYPES
AND THEIR STRUCTURES [7]

D-Types

Column	Column Name	Contents	Description
0	@ID	*Name of the dictionary entry*	Identifies the field
1	Type	D	Type code
2	Loc	*Field number*	Location of the field in the data table
3	Conv	*Conversion code*	Formula to convert the data into external format
4	Name	*Column heading*	Name used as column heading
5	Format	*Width and justification*	
6	SM	S | M	Single-valued or Multi-valued flag
7	Assoc	*Phrase name*	A phrase name that links multi-valued fields
8	Data Type	*Data type*	SQL data type if present (Usually not)

I-Types

Column	Column Name	Contents	Description
0	@ID	*Name of the dictionary entry*	Identifies the field
1	Type	I	Type code
2	Exp	*I-Type expression*	Expression that produces values for the field
3	Conv	*Conversion code*	Formula to convert the data into external format
4	Name	*Column heading*	Name used as column heading
5	Format	*Width and justification*	
6	SM	S | M	Single-valued or Multi-valued flag
7	Assoc	*Phrase name*	A phrase name that links multi-valued fields
8	Data Type	*Data type*	SQL data type if present

A-Types & S-Types

Column	Column Name	Contents	Description
0	@ID	*Name of the dictionary entry*	Identifies the field
1	Type	S | D	Type code
2	Loc	*Field number*	Location of the field in the data file
3	Name	*Column heading*	Name used as column heading
4		C;number[;number …]| D;number	Links multi-valued fields
5		NONE	Not used
6	Data Type	*Data type*	SQL data type if present (Usually not)
7	Conv	*Conversion code*	Formula for converting stored data to external format
8	Corr	*Correlative code*	Formula that produces values for a field
9	Typ	L | R | T |U	Justification code for a field
10	Max	*Width*	Column width for a RETRIEVE report