A Prototype for Shell/Database Integration

Ben Uphoff, Marc Dreyfuss

Abstract

As database technology becomes more and more commonplace, it is beginning to become feasible to integrate this technology into the core operating system functionality. In the past there have been two key elements that have kept database technology out of the realm of operating system (OS) functionality. First, database management systems (DBMSs) have traditionally been very expensive. Second, DBMSs have typically been difficult to use for many users. Once these problems are addressed, DBMS functionality can be integrated into the OS.

The first problem, the prohibitive cost of DBMS, has been addressed by the popularity of open source development and the Internet. As the DBMS market has matured, several powerful products such as PostgresSQL and Oracle 8i have become available at no cost.

The second problem, the ease of use, is the focus of this project. We have developed a prototype shell interface that attempts to provide a consistent, straightforward interface to three DBMS products: Sybase, PostgresSQL and Minirel. These three products all have differing syntax and execution parameters. Our prototype hides these differences as much as possible without sacrificing any functionality. Thus, a user can connect to and operate on a Sybase database in the same manner as he or she would on a PostgresSQL database. This transparent operation is the key to the prototype.

1. Problem

Database technology has traditionally been very useful but expensive and hard to use. The popularity of open source software development and Internet has made robust DBMS products freely available to the public. Unfortunately, these systems retain their traditionally difficult learning curve.

Every DBMS differs in some way, either in syntax or application program interface (API). Some systems use Structured Query Language (SQL) while others use a proprietary grammar for communicating with the DBMS. The syntax of SQL, a supposed standard, can also change from product to product. The APIs of various products can be very complex and do not always meet the needs of the user. Some products do not provide an API at all.

If a user only had to learn one particular DBMS, many of the difficulties of learning the product can be overcome through time and training. However, it is often necessary in the business world for a user to learn several DBMS products. This makes it hard to keep track of what syntax works with what system and what API calls to make when writing code. Our solution attempts to alleviate as much of this difficulty as possible without limiting the power of the DBMS.

2. Solution

The logical point of integration on Unix is the shell. Although many OSs today are graphical and do not rely on a shell, Unix users still spend a great deal of time in the shell. Furthermore, the three products that we evaluated, PostgresSQL, Sybase and Minirel, all used the shell as their primary user interface.

The three key functions that have been added to the shell are:

Automated connection and log in
Automated data loading
Command line query execution

2.1 Connection

The prototype allows seamless connection to various DBMS systems. The user must define a connection script that provides information such as log in credentials, location of the DBMS. The script must also define any environment variables needed by the DBMS and start any background processes. The user must also define a disconnection script to log the user off and clean up the environment.

Once the connection and disconnection scripts are created, the user does not need to worry about the connection details. If the user exits the shell or connects to a different DBMS, the disconnection script is run. The connection script can be run at anytime by entering *db connect dbname*. Where dbname is the name of the DBMS to connect to. To disconnect the user simply types *db disconnect*.

It is important to note that the shell can only be connected to one DBMS at a time. This was a conscious decision to simplify the various commands used to invoke DBMS functionality. Each call to a database command would have required the database name to be given if multiple connections were allowed. We felt that the case where a user would need multiple connections would be rare and thus always typing the database name would seem useless. The user can overcome this limitation by starting a new shell and connecting to a different DBMS there.

2 1 Command Execution

The shell provides users the ability to execute commands from the shell. Most DBMS products have some sort of user interface where commands can be run against the DBMS. This functionality has been added to the shell to avoid the user having to run product specific program to execute queries and such. For instance, in the shell prototype a user can enter *db command select * from Students* to get a listing of the Students table. This command is sent on to the DBMS without modification. This is the one area in the system where the user must know the particular syntax of the DBMS. The motivation for this was to avoid limiting the type of commands that can be executed from the shell. By using this interface, the user can do anything with the DBMS that can be done from the normal user interface but without ever leaving the shell environment.

2 2 Data Loading

The biggest benefit of this prototype is the power and flexibility of the data load functionality. The prototype provides users with the ability to redirect the output stream of a program or a file to a device called *db*. When the shell detects the redirection, the output is parsed to conform to a user-defined schema. The schema contains information on the layout of the output stream as well as the table layout in the DBMS.

Once parsed, the data is inserted into the DBMS. The user is then free to query the data using the command execution procedure outlined above. Any information that did not parse is placed in an error file. The user can then look at the error file and make adjustments to the schema or the output.

The key to making the interface seamless is the schema definitions. These schemas can be created in three ways: using a graphical user interface, using a shell program or hand coding. The schema definitions are stored as extensible markup language (XML) files. This makes the schemas easy to read and edit by person or program. It also allows developers to easily create schema definitions for their programs.

The schema definition stores detailed information about the formatting of the output or file. If the data does not have a consistent format of some sort, no schema definition can be created for it. The schema definition can have any number of fields each with length information, type information and delimiters. Delimiters can be mixed through out the schema.

The schema definition is detailed enough to describe any consistently formatted output with two minor drawbacks. The first restriction is that fields that may contain their delimiter must be of fixed width. Building a parser to overcome this problem is out of the scope of the prototype.

Secondly, the prototype enforces a strict one line to one record correlation. Thus, the schema cannot define data that spans more than one line. This is a weakness in the parser that can be addressed without major changes in design. However, we felt that this case was rare enough to not merit redesigning the parser.

Aside from these two minor flaws, the schema definition is a robust representation for single line data. Once a schema definition is created for a particular set of data, the user never has to worry about how the data is populated into the DBMS. All the user has to do is redirect the data and then check the error file to make sure the necessary data was inserted.

3. Implementation

There are two layers in the prototype design. The shell layer is responsible for communicating with the shell and executing various commands. The DBMS layer is responsible for deciding how the shell layer should go about communicating with a particular DBMS. These layers are necessary to provide a level of indirection between the various types of shells and DBMS products.

Between the shell and DBMS layers is the core functionality of the prototype as well as various code modules, called engines. The core module contains the XML parser for parsing schema definitions. It also contains the necessary logic for extracting a particular field value from a redirected output stream. The engines are code modules that generate scripts for creating tables and performing inserts. There is one engine for each supported DBMS as well as a generic SQL engine that can be used on a DBMS that adheres to the SQL standard.

3.1 The Shell Layer

As stated above, the shell layer is responsible for communicating with the shell. This layer of indirection allows the prototype to be easily ported to other shells such as Bash or Bourne.

The first communication responsibility that the shell layer has is to handle to connection and disconnection calls. The command *db connect dbname* signals the shell layer that a connection needs to be handled. The shell layer first looks for a connection script for *dbname* in a preset directory. The shell layer must also check to see that the script is for the proper shell type. Once the right script is found, the shell layer checks to see if the shell is currently connected to another DBMS. If this is the case, the disconnection script is run for that DBMS. Finally the connection script is run. Similarly, a call to *db disconnect dbname* will find the proper disconnection script and execute it.

The second communication responsibility of the shell layer is to execute commands passed into it by either the core module or the DBMS layer. Either of these layers can send the shell layer a command. The shell does not need to worry about the command aside from checking that the shell is connected to a DBMS. Once this check is made, the command is executed.

The final responsibility of the shell layer is to monitor errors generated from the commands that it executes. This functionality is not implemented by our prototype but is discussed in section 5.1.

3.2 The DBMS Layer

The DBMS layer can communicate with the shell layer, the core module and the engines. This layer provides a single point of interaction for all communication with the DBMS. Although the DBMS layer decides what calls are needed to perform a specific task, it leaves the execution of a command up to the shell layer. Again, this is so the DBMS layer does not have to worry about the details of the shell environment.

The DBMS layer is used when performing data loading operations. The core module will instruct the DBMS layer to generate a table creation or insertion script based on a given schema definition for a set of values. The DBMS layer takes the information provided by the core module and passes it to the engine for the connected DBMS. The engine returns a command that the DBMS layer then sends to the Shell layer for execution. This is the simplest element in the system; it does little more than handle the flow of traffic in the system.

3.3 Engines

Connected to the DBMS layer are a number of code modues, called engines, that generate product specific commands or scripts. Every supported DBMS needs to be tied to an engine. A generic SQL engine is provided, as are engines for Sybase, Minirel and PostgresSQL. The following table is an example of a table creation script and an insertion script for a schema definition by three different engines:

Engine	**Create**	**Insert**
SQL	`CREATE TABLE lsl(FileType CHAR(10), OwnerPerms CHAR(3), GroupPerms CHAR(3), OtherPerms CHAR(3), OwnerID CHAR(8), Size INTEGER, ModifyDate CHAR(6), ModifyTime CHAR(5), Name CHAR(255))`	`INSERT INTO lsl(FileType, OwnerPerms, GroupPerms, OtherPerms, OwnerID, Size, ModifyDate, ModifyTime, Name) VALUES ('File', 'rw-', 'r--', '---', 'bduphoff', 1917, 'May  8', '10:21', 'SQLEngine.class')`

PostgresSQL	`CREATE TABLE lsl (FileType VARCHAR(10),`
	`OwnerPerms VARCHAR(3),`
	`GroupPerms VARCHAR(3),`
	`OtherPerms VARCHAR(3),`
	`junk VARCHAR(1),`
	`OwnerID VARCHAR(8),`
	`Size DECIMAL(8),`
	`ModifyDate VARCHAR(6),`
	`ModifyTime VARCHAR(5),`
	`Name VARCHAR(255) );`
		`INSERT INTO lsl (FileType, OwnerPerms, GroupPerms, OtherPerms, junk, OwnerID, Size, ModifyDate, ModifyTime, Name) VALUES ('File', 'rw-', 'r--', '---', '1', 'bduphoff', 1917, 'May 8', '10:21', 'SQLEngine.class');`

Minirel	`create lsl(FileType=s10, OwnerPerms=s3, GroupPerms=s3, OtherPerms=s3, OwnerID=s8, Size=i, ModifyDate=s6, ModifyTime=s5, Name=s255);`	`insert lsl(FileType="File", OwnerPerms="rw-", GroupPerms="r--", OtherPerms="---", OwnerID="bduphoff", Size=1917, ModifyDate="May  8", ModifyTime="10:21", Name="SQLEngine.class");`

The differences in the above example illustrate the need for the various engines. By its name, PostgresSQL would suggest that it would run the first create table script but this is not the case. PostgresSQL requires several syntactical elements that SQL does not use. The PostgresSQL engine was easily created by extending the existing SQL engine. This level of modularity allows the prototype to quickly switch between various database products. It also allows for new products to be added to the prototype. There are only two steps in adding support for a DBMS. First, the connection and disconnection scripts must be created. Then the new engine must be created, usually by modifying an existing engine. Without the modularity of the engines, this process would be much more complicated.

3.4 Schema Definitions

The area of most complexity in the prototype is in the schema definitions. As stated above in section 2.2, the schema definitions are the critical element in the data loading procedure.

Schema definitions are created by the user to map program output into a relational model. A schema definition is stored as an XML data set that conforms to the Schema Definition document type definition shown here:

```
<!ELEMENT SchemaDefinition (Field+)>
<!ATTLIST SchemaDefinition name CATA #REQUIRED>
<!ELEMENT Field (Value+)>
<!ATTLIST Field name         CDATA #REQUIRED>
<!ATTLIST Field isVariable   CDATA #REQUIRED>
<!ATTLIST Field minInLength  CDATA #REQUIRED>
<!ATTLIST Field maxInLength  CDATA #REQUIRED>
<!ATTLIST Field maxOutLength CDATA #REQUIRED>
<!ATTLIST Field delimiter    CDATA #REQUIRED>
<!ATTLIST Field type         CDATA #REQUIRED>
<!ATTLIST Field isIncluded   CDATA #REQUIRED>
<!ELEMENT Value EMPTY>
<!ATTLIST Value inValue  CDATA #REQUIRED>
<!ATTLIST Value outValue CDATA #REQUIRED>
```

This is a very convenient way to store such data sets. There are a variety of XML parsers available for both Java and C++. This prototype uses IBM’s XML4J, a validating XML parser for Java. Very little coding is required to build objects from the XML. XML is also easy to read and edit making schema definitions easy to modify and debug by the user.

However, storing these data sets has some drawbacks. First, using a XML parser is slower than reading fields from a flat file. We feel that this tradeoff was acceptable, as this is a one-time cost when the schema is first loaded into memory. These schemas could also be cached to reduce this delay. Also, storing a file as XML takes up somewhat more space than a flat file might. Again, we felt that this was not a significant factor as these files are not likely to be over 10K in size.

The primary function of the schema definition once in memory is to extract fields from a piece of data. The method to do this, *getFieldValue()*, has two parameters: the field index and the string to extract the value from. In our implementation the string was always a single line from the program output file that was being mapped. This restriction could be relaxed to provide support for multi-line schema definitions. The return value is the value of the field as a string. In any case where the field value could not be processed, a null string was returned. This signals the object calling *getFeildValue()* that the line could not be processed and should be placed in the error file.

The first step in *getFieldValue()* is to handle value mapping. Value mapping can be added to a field to map an input value into a different output value. An example of this might be a field in the program output that was 1 for a true value and 0 for a false value. The user might want to see "true" instead of 1 and "false" instead of 0. The schema definition allows users to define such relationships. *getFieldValue()* checks to see if a field has any such values and does a lookup on the value it found. If there is no match in the value mapping, a user defined default value is used.

Before returning a value, *getFieldValue()* makes several checks on the field value it found. First, it compares the length of the field to the length specified in the schema definition. If the field is too long or too short, a null is returned. Next, the type of the field is checked. A null is returned as in the previous cases.

The schema definition also has several utility methods for returning the contents of the schema definition including fields and value mapping attributes. One other notable method is *validate()*. This method checks the state of the schema definition for errors or inconsistencies and returns a code describing the error. This method is used in the SchemaDefiner tool to validate user input.

4. Benefits

The prototype described here offers several important benefits to the user. The most notable are:

- Seamless data loading
- Automated login/switch DBMS
- Makes database technology more assessable

4.1 Data Loading Benefits

Seamless data loading is of key importance because it reduces the difficulty of populating the database. Users are often turned off by database technology because of the pain of data entry or the complexity of product specific importing tools. Our prototype alleviates these problems by providing a consistent, easy to use interface. We chose the common redirect operator because most Unix users are familiar with the concept of redirection. The only difficult part of the data loading process is setting up the schema definition.

The benefits of this form of data loading can be illustrated by the problems faced by scientists. Scientific research is often done as a collaborative venture spanning corporations and academic institutions. Ultimately, it is useful for the researchers to funnel their data into a central location for analysis. This can be difficult as data can exist in many different formats and locations. Our solution helps streamline the process by allowing researchers to tie a schema definition to their data and then populate a central database using the simple redirect operator. Thus, researchers are not bound to use one specific tool to generate research data. They can use a variety of tools and still easily populate the central database.

4.2 Benefits of Automated Connections

The benefits of an automated connection and disconnection procedure are numerous. First, there is no need to set environment variables or start background processes. This is all done for the user in the predefined connection script described in section 2.2. Second, when switching to a different DBMS, there is no need to disconnect from the current DBMS. This is all done automatically. The last benefit of the automated connections is the level of flexibility provided. The connection and disconnection scripts can be customized for local or remote connection. The user, or perhaps an administrator, has complete control over these scripts and can configure them for any sort of installation. These elements allow the user to spend more time working with the data and less time working with the ugly side of the DBMS.

4.3 The Overall Benefit: An Accessible Environment

The overall benefit of the prototype is that it makes database more accessible to the typical user. The prototype tries to hide the messy details of the particular DBMS product where the user cannot see them. This makes for a friendly, easy to use environment. We feel that such an environment will allow users to use database technology where they might otherwise have thought it to be too much trouble.

5. Performance

The nature of our prototype makes it difficult to measure performance. Our goal was to provide a simplified mechanism for using various DBMS products. To achieve this, we have created two layers of indirection to provide portability across different shells and DBMS products. These additional layers mean more overhead when performing database operations using the *db* command. Thus, performance will always be somewhat slower when running though the shell as opposed to using a product specific interface.

We feel that this trade off in performance is acceptable when the easy of use benefits discussed in section 4 are considered. In our experiences with the prototype, we found it much more desirable to execute commands from within the shell using the *db* command instead of running from the DBMS's interface. This allowed us to stay in a true shell environment at all times. This became even more convenient when we were switching back and forth between several DBMSs. It was clear to us after working with the prototype that it was a clean and easy to use interface. We also found that the data loading mechanism worked very well for many data sets.

Although performance is hard to measure for the prototype, we have made an attempt at doing so. We provide two areas of performance comparison: raw execution and usability performance.

For the raw execution tests, we attempted to find the cost for an operation to traverse the layers added by the prototype. In this test, a simple program was created to generate a table creation script for PostgresSQL and an insertion script of ten rows. The test program used the same code to generate the scripts as the prototype. Next, the table creation and insert scripts were run by PostgresSQL's psql interface. The total time of these values is representative of the parsing time plus the actual cost of creating the table and performing the inserts. The same data set was then redirected to /dev/db to calculate the execution time from within the prototype. The difference of the two runs shows the general overhead added by the layers of indirection for this case. The following table summarizes the results. All execution times that are averages of five consecutive runs on a clean database.

# of Inserts	Raw Execution (seconds)	Prototype Execution (seconds)	Differential (seconds)
10	3.06	7.598	4.538
100	7.726	9.558	1.832
1000	59.482	11.372	-48.11

The results of this test appear inconsistent at first. The prototype appears to beat raw execution for larger inserts. This is because the prototype intercepts the majority of the output to the screen. Because the prototype does not display some messages, it actually executes mush faster than running through psql. This is an added bonus in our design. The set of ten runs does illustrate the overhead of running in the prototype however. Running through the prototype is over twice as slow in this case. We expect that the 4.5-second differential would be constant if the prototype and psql displayed the same number of messages.

Our second set of tests shows how the prototype functionality could be used to speed up certain common tasks that a system administrator might perform.

Find vs load then query

6. Future Integration

As previously stated, this implementation was designed as a prototype to show how database functionality could be integrated into a shell environment. We chose Java as our coding platform because of the numerous available classes for managing processes and string manipulation. The String class in Java proved invaluable for creating a parser. We ran into difficulty with the Process class and its supporting classes however. We found it easy to send a command to the shell but found it hard to monitor the process's progress.

Another motivation for using Java was to avoid getting bogged down in the complexity of a shell like tcsh. We felt that our focus should be on functionality and ease of use and not in a true integration. This turned out to be a bad decision because Java did not give us the control over processes that was needed to make a well-performing prototype. We would have been better served to work in C and take advantage of UNIX system calls like *fork()*. This proved most costly in our attempt to support Sybase. Commands sent to isql, the shell interface to Sybase, never seemed to return or resulted in cryptic errors. If we had used the C libraries for Sybase in conjunction with Unix system calls we could have successfully supported Sybase.

A future attempt at integration of our prototype would require the entire system to be rewritten in C and added to a shell like tsch. This would provide numerous advantages including better overall performance and superior control of processes and the shell environment.

Aside from the inherent performance benefits of C over Java, the biggest gains in performance would come from using the DBMS APIs instead of simply issuing commands to a shell process. Most DBMS products have some sort of API routines written in C. These calls are fast and provide the calling process with greater control over error conditions and conflict resolution.

By using commands like *fork()*, the shell would have complete control over API calls and connection and disconnection scripts. This would allow the system to detect when a DBMS went off-line and allow the user to respond to the event. Our prototype has no knowledge of the status of a DBMS due to the limited functionality provided by Java. This functionality would be critical in a true implementation.

Another addition would be the /dev/db device. This device does not really exist in our prototype but should be implemented as a UNIX device in a future implementation. Much of the prototype's functionality could be placed in this device, minimizing additions to the kernel.

One last addition to a future integration is the reduction of I/O in data loading. Currently, the prototype bundles inserts into a file before sending them to the DBMS. This is effective for large data sets but unnecessary for small batches of inserts. Ideally, inserts should be run in small, memory resident batches for both large and small data sets. This will reduce the cost of I/O to reading the program output in a single pass.

7. Conclusions

In the end, our prototype shows that an easy to use database interface can be designed for a multiple database environment within the shell. We have taken away many of the dirty details of using a DBMS by automating connection and disconnection and providing a seamless interface for data loading. The user defined schema definitions provide the users with a robust but not overly complex means of representing formatted program output in a relational manner. In all, these enhancements to the shell combine to make using database technology more appealing to the user.

However, we acknowledge that our prototype has some drawbacks. As stated in section 6, the prototype needs to be written in C to provide optimal performance and control over the shell environment. Also, because of the two layers of indirection, DBMS and shell, the prototype can never be expected to be faster than execution from a product specific interface. These layers are essential however as they allow the prototype to be ported to other shells and DBMS products with minimal effort.

An interface such as this would be a useful addition to shell functionality in any operating system. We encourage other researchers to consider this area for future study.