TeX and Databases – TeXDBI

K. Anil Kumar
Linuxense Information Systems Pvt. Ltd.
Trivandrum, India
http://www.linuxense.com
anil@linuxense.com

Abstract

Report generation is one of the demanding areas of enterprise computing. It involves complex database queries, drawing conclusions, making projections and presenting them in an easy-to-comprehend manner. This paper describes how to use TeX to deal with the presentation aspects in a productive way.

Introduction

TeX is highly programmable and has a high degree of consistency in maintaining the structure and layout of the documents generated. Moreover, no other typesetting system can claim the precision control of typesetting parameters that TeX does. This makes TeX a good tool to prepare reports both for printing and for Web publishing. However, TeX lacks one capability as a report generation tool: it cannot directly communicate with external systems like database engines. Mostly data for reports reside in database systems and must be retrieved based on rules. So we are compelled to depend on other technologies and systems to achieve this. This may also mean bringing in additional people to get things done.

First, let us have a look at various techniques for generating reports using TeX which are in use today.

Ways to Generate Reports Using TeX

We have been using TeX to generate reports using a variety of techniques. And these techniques can be classified into two categories: 1) generate TeX documents using other languages; 2) embed other languages in TeX documents and use a preprocessor. Both of these techniques are explained below.

Use Database-enabled Languages to Generate TeX Documents A Java (for example)[1] program can create a TeX file with the data retrieved from a database. We then compile this TeX code to get a presentable report. The process contains the following steps:

[1] The language could be Perl, C++ or any other with the capability to interact with a database engine.

1. A TeX programmer prepares a TeX document that we will call a template. This template contains no useful information itself, though it is a complete TeX document as far as TeX is concerned. We add the specific data to it, e.g., it may be a table construct defined with all the required parameters, but no actual rows of data. The template only becomes a useful document when the data is added.
2. This TeX template is handed over to the Java programmer to write a program, with this template hard-coded within it, which will in turn write a TeX document with the data retrieved from a database engine.
3. The generated TeX document is compiled to get the report.

There is one disadvantage with this approach. It is difficult for a TeX programmer to intervene to make a change in the report format (i.e., in the template) after step 2 given above. This is because the TeX code has already been embedded into the Java program in a very different form, which doesn't make any sense to a TeX programmer at all. There are only a couple of solutions to tackle the situation:

1. Repeat step 1 and 2 with the modified format specification; or
2. Have the TeX programmer and the Java programmer work together on making the required changes.

In a production environment, where work must be done very quickly, neither of these approaches are feasible. To repeat the process all over again is definitely a bad idea and bringing two groups of people, as suggested in the second solution, to work on a problem is difficult to manage and time-consuming.

Embed Other Languages in TEX There is a better approach than this first one. We can embed statements in a database-capable language in the TEX template. Then through a pre-compilation process we can execute those embedded code snippets and replace them with the data retrieved from the database to make a "data-filled" TEX document.

Let us analyze the steps involved in generating a report this way.

1. As before, a TEX programmer prepares a template.
2. This template is handed over to the programmer who will embed code snippets required to retrieve data from the database and fill-in the template.
3. The template undergoes pre-compilation and the resulting TEX document is compiled. The report is ready.

In this approach, even after embedding the code snippets the TEX template will look like a regular TEX document and a TEX programmer can still modify it if there is a specification change. So there is no need to bring in the other programmer to make a modification in the report presentation.

Pre-compilation is the major disadvantage of this approach. Every time to generate a report one has to pre-compile the document and then run the TEX compiler to get the report. Moreover, the TEX coder either has to learn a database capable language or he/she has to depend on someone who knows it.

Enable TEX to Communicate with External Systems This approach will enable TEX to communicate with external systems like database engines to retrieve information to typeset reports directly. There is no need to depend on other languages and there is no pre-processing involved. This approach has two advantages:

1. A TEX programmer can prepare reports by himself/herself without depending on another language programmer.
2. Development cycle is faster. Changes can be incorporated in the TEX document directly and run the TEX compiler; and it is done!

How to Enable TEX to Talk to Database Engines?

A running TEX compiler is a process[2]. For a process, communicating with the external world means communicating with other processes, either running on the same computer or on a different computer in the network. TEX has no built-in interprocess communication facility but they are commonly provided via the following mechanisms:

1. Writing to a file using `\write16`.
2. Executing a shell command using `\write18`.
3. Reading in a file using `\input`.
4. Writing to terminal using `\write15`.

We are limited to these mechanisms for I/O capabilities with TEX.

Any modern operating system supports named pipes[3] and sockets for communication between two arbitrary processes. Files are also a way of passing data to the external world. However, considering the I/O capabilities of TEX, communication with sockets does not seem feasible and so the option is either files or named pipes.

Files or Named Pipes? We can make TEX to write to a normal disk file and instruct an external system to read from it. This is a simple way of exchanging data between two arbitrary processes. It is quite simple to understand, and very easy to implement. But the downside is that it is very difficult to synchronize communication and it can become out of control in certain situations. Named pipes or FIFOs are better in this area.

Named pipes, also known as FIFO structure, can be effectively used with TEX for interprocess communication. For TEX, a named pipe will appear as a regular file. When TEX writes to this file, the operating system stores the data in a buffer and then it can be read by the other program. For the reading program, data will appear in the order it was written. Also the reading process will be blocked till the other party writes to it. Similarly, the writing process will be blocked until the other process starts reading it [4]. This will provide the required synchronization for the "conversation."

Hence TEX can write and read from a named pipe to talk to another process and that process can do the same. Thus, it becomes possible for TEX to communicate with another process and the architecture discussed in this paper for database communication is built upon this idea.

[2] That is, an operating system process which can be identified by a process ID.

[3] A named pipe, also known as FIFO (first-in-first-out), is similar to device files in Unix/Linux. They have a *disk inode* (and a file name) and hence can be accessed by any process. As the name implies, with FIFO the first byte written into it will be the first byte read from it.

[4] Here we assume that the named pipe is opened in *blocking* mode. If the named pipe is opened in *nonblocking* mode the read will return whatever bytes are available (perhaps none).

A TEX Abstraction for Database Communication: A Middle Layer

Interprocess communication and named pipes are far beyond the interest of a TEX programmer and it is definitely a bad idea to suggest a TEX programmer to use all these things while doing TEX coding! So a good implementation of this idea should provide a TEX-like abstraction of the mechanisms discussed above. This abstraction should have the least possible learning curve and should adhere to standard TEX coding conventions.

An n-Tier Architecture Here we are going to make TEX to talk to a database engine. At the top there is the TEX compiler and at the bottom there runs a database engine. And in between we introduce two components: one component gives a TEX abstraction of the systems beneath and the second component bridges the TEX abstraction and the database engine. These two components together are called TEX-DBI.

TEX Abstraction of a Database Operation

Querying a database involves opening a connection to the database, passing an SQL query, retrieving the result and finally closing the result object and then the database connection. A LATEX package can be used to define a set of macros to do all these jobs on behalf of the TEX programmer.

Following are the macros that we defined in the implementation of this idea.

```
\begin{tex-dbi}[host=xx.yy.zz,%
  dbname=sampledb,uname=anil,passwd=mypw]
```

This will initialize a database transaction for the session[5]. There can be more than one transaction per session but they are not supposed to overlap. Here is a description of the parameters:

Parameter	**Meaning**
`host`	Name of the computer running the database engine
`dbname`	Database name
`uname`	Database username
`passwd`	Database password for the specified username

```
\texdbiexec{"select * from employee"}
```

This macro passes the SQL statement to the underlying database engine. The TEX macro is not responsible for checking the accuracy of the SQL given; it just passes the statement to the underlying system.

[5] Or current pass of compilation.

```
\texdbicount{}
```

This macro returns the number of rows (possibly zero) resulting from the query. This value may be used as the limiting value for a loop or just to check whether the query returned successfully.

```
\texdbinext{}
```

The result object exposes only one row at a time of the possibly several rows returned by the query. This macro moves an imaginary pointer through the result set, making each row current in turn. When a new result is obtained the imaginary row pointer doesn't point to any row; a call to this macro then sets the pointer to the first row returned.

```
\texdbivalue{field_name}
```

This macro returns the specified field of the current row. `field_name` should match the name of the field specified in the SQL statement[6].

```
\end{texdbi}
```

This macro ends a database session. The connection will be closed and the underlying component will be ready to start a new transaction.

Normally, these macros are called in the sequence given above. `\texdbinext{}` can be called as many times as the number returned by the macro `\texdbicount{}`. `\texdbivalue{}` can be called with appropriate parameters as many times as required after each call to `\texdbinext{}`.

The Bridging Component The TEX macros described in the above section communicate with this component through a named pipe as described earlier. It is called a bridging component because it connects TEX and the database engine. The communication between the bridging component and the database engine itself happens through a TCP/IP or Unix domain socket.

The request in the TEX code will be passed to the bridge by the macros through the FIFO and these requests are translated to equivalent JDBC or ODBC commands (or in some other format required by the mechanism being used by the bridging component) as the case may be. The results from the database engine obtained by the bridge is written back to the FIFO for the TEX macros to read.

[6] This rule is enforced by the underlying bridging component which makes use of JDBC, ODBC or similar mechanisms to transact with a database engine, as described next.

Adding Session Capability to Make the Middle Layer a True Multi-user System

It is also possible for this system to support sessions, thus making it a true multi-user system. A Web server session-ID-like mechanism can be employed to identify each request and thus to communicate with multiple TEX compilation sessions simultaneously.

An alternative to this session-ID approach is to use session-specific FIFOs: TEX macros can create a FIFO through a shell command and the name of the FIFO can be passed to the bridging component as part of the transaction initiation request. The rest of the transaction can happen though that FIFO.

If multiple concurrent sessions are supported, only one middle layer is enough to support multi-user enterprise needs.

Conclusion

Enabling TEX to communicate with a database engine will eliminate the need to involve other languages to generate quality reports. A TEX programmer can easily learn and use the TEX macros defined by the TEX-DBI system. Development life-cycle will be shorter and the process will be completely in the hands of a TEX programmer.

An implementation of this concept can be found at: `http://www.linuxense.com/oss/texdbi/`.

References

[1] Knuth, Donald E., 1986, *The TEXbook*, Addison-Wesley.

[2] Bovet, Daniel P. & Cesati, Marco, 2001, *Understanding the Linux Kernel*, O'Reilly.