Dependency theory e-learning tool.

Paul Douglas
Cavendish School of Computer Science, University of Westminster

Steve Barker
Department of Computer Science, King's College, London

Dependency Theory E-Learning Tool

Paul Douglas
University of Westminster, London, UK
P.Douglas@wmin.ac.uk

Steve Barker
King's College, London, UK
steve@dcs.kcl.ac.uk

Abstract

In this paper, we describe an e-learning tool that we have developed to assist University students studying Relational Schema design. The tool employs Expert System techniques to create a learning environment in which students can explore the concepts of dependency theory, and the normalization process. The tool is able to respond in an individualistic way to student input and allows students to construct their own learning environment and develop their understanding of the material at a pace that is controlled by the individual student. Our formative and summative tests indicate that the tool provides students with a different and valuable type of learning experience when compared with a traditional, textbook-based approach.

1. Introduction

We describe a learning tool that we have developed and used to help University-level students to learn certain essential notions in database schema design, specifically within the area of the so-called *normalization* [13] process, which is based on the underlying concept of *dependency theory* [22]. We regard the learning tool as a piece of intelligent software, where the term "intelligently" is interpreted by us as the capability of responding to a student's self-selected input by detecting, diagnosing and explaining his/her errors or confirming that his/her understanding is correct.

We have found that students often find the theoretical concepts of dependency theory difficult to assimilate. Traditional textbooks can offer very good coverage of the material, but little scope for students to test their own understanding through a process of practical experimentation. Many textbooks provide no practical exercises but, even where they do, they are necessarily limited in scope, and do not cover a sufficiently large range of levels of difficulty. Some textbooks that do provide exercises do not give solutions, so students cannot determine whether they were able to correctly solve the problems: this is not a very useful learning experience.

Some courses that teach Relational Databases take the approach that the use of a schema design tool will almost always deliver a schema that is in *third normal form (3NF)* [13], and that teaching dependency theory is not really necessary (see, for example, [3]). However, we disagree with this point of view. There are many aspects of relational database technology that are directly related to the data dependencies that exist within a database, and students cannot properly consider these issues without a proper understanding of dependency theory. The use of design tools also suggests that a relational database schema has been somehow finalized once it has been put into third normal form, leaving students with even less understanding of normal forms beyond 3NF. Students will not be able to critically evaluate alternative designs, or make informed choices about levels of normalization, if they do not understand the principles upon which design decisions have been based. Functional dependencies are also important for a proper understanding of the concepts of *candiate keys*, *superkeys*, constraints, and the theoretical foundations of relational database systems.

The remainder of this paper is organized as follows. Section 2, gives a brief overview of dependency theory and its relationship to schema design. Section 3, covers the design of the user interface to the software, and outlines a sample user session. Section 4, covers normalization algorithms, and describes our approach for implementing the software. Section 5, discusses our evaluation of the software. Finally, conclusions and suggestions for further work are given in Section 6.

2. Dependency Theory and Schema Design

Databases are now widely employed in a variety of applications, and many students of Computer Science will study database systems as part of their course. The Relational Data Model [4] remains the most widely taught data model, and the design of relational databases is a core component of the computer science curriculum.

Poorly designed relational databases may exhibit *redundancy* and *update anomaly* problems [4]. To minimize the

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)
0-7695-2108-8/04 $ 20.00 © 2004 IEEE

effects of these (and other) undesirable features, relational schemas are usually refined through a process of *normalization*. [5] and [6] introduced, respectively, second (2NF) and third (3NF) normal forms, based on *functional dependencies* (FDs). A FD $X \rightarrow Y$ between two sets of attributes X and Y will apply to a relation R if and only if the following criterion holds: for each pair of tuples t_1 and t_2 in the instance of R, whenever t_1 and t_2 agree on the value for X then t_1 and t_2 agree on the value for Y.

Boyce-Codd normal form (BCNF) is the highest normal form based on FDs. However, 3NF is often easier to achieve, and is frequently employed in practical implementations (not least because *dependency preserving decompositions* into 3NF are always achievable, but not with decompositions into BCNF). 3NF is usually the level of normalisation achieved by design tools. [9] introduced the concept of *multivalued dependencies* and *fourth normal form (4NF)*. [18] introduced *join dependencies*, and [10] described an associated normal form, *fifth normal form (5NF)*.

A much fuller treatment of the material on dependency theory can be found in, for example, [8].

3. Interface Design

3.1. Our Development Methodology

Our approach to developing our learning tool for database design initially involved us adopting a *phenomenographic* method [16] for information gathering on students' understanding of concepts in dependency theory. By conducting 'dialogue' sessions with students we identified the strategies students used to understand the basic concepts. From our review of the notes taken at the dialogue sessions, we were able to develop a prototype system for supporting students in learning about dependency theory.

As the software evolved, we made increasing use of Gagne's *event-based model of instruction* [11] to decide what material a user of the tool should be offered and the order in which information ought to be presented to a learner. Following [11], prominence is given to the distinctive features that need to be learned, different levels of learning guidance are supported for different types of learners, informative feedback is given, and learning takes place in a student-centred, interactive way, but with support available to students as and when they need it.

The interface is intended to be simple to use; it is menu-based and all data entered is case-insensitive. Users are prompted throughout a session for the correct data to enter, and can return to the main menu at any time. It is possible to enter multiple schemas, save them, and return to them later within a session. Sessions can also be retained in a file and can therefore be suspended and resumed.

3.2. A Learning Session

Users invoke the software by using the Java JRE. On a unix system, this is started by the *java* command, so a session on a typical system would start thus:

```
jaguar% java ~cspub/java/schemas
```

whereupon the system will respond with the opening menu:

```
                MAIN MENU
1 Enter a Schema
2 Help
3 Exit

Enter Choice (1-3):
```

The "help" option gives some general guidance on how to use the system; the "exit" option terminates the program. Normally at this stage, the user will enter a schema. The system will first prompt the user to enter the names of the attributes:

```
Enter attribute names, using spaces to separate them.
Names must be single chars or character strings only:
```

to which the user will typically respond with something like:

```
a b c d e
```

The next step is for the functional dependencies to be entered. The user is first prompted to enter a determinant, then its dependent attributes. The process will be repeated for each determinant. The system loops around these input processes until a blank line is entered: for each determinant the user is repeatedly prompted to enter another dependent attribute until a blank line is entered; the user is then invited to enter another determinant, and this process in turn repeats until a blank line is entered. An example is:

```
Enter a determinant
If multivalued, use spaces as separators:

a

Enter a dependent attribute
If multivalued, use spaces as separators: b

Enter another dependent attribute
(return to end): c

Enter another dependent attribute
```

(return to end):

Enter another determinant
(space to end)
. . .

and so on until the process is complete, when the system will respond with a display of the information entered and another menu:

Your schema has attributes:
[a,b,c,d]
and FDs:
[a]–>[b]
[a]–>[c]
[c]–>[d]

Choose an option:
1 3NF Decomposition
2 Help
3 Exit

Enter Choice (1-3):

The "exit" option returns the user to the previous menu; the "help" option gives some information about the decompostion process.

The user will normally select one of the decompostion options; an example of the output if 3NF is chosen follows:

Finding a minimal cover. . .
. . .checking right reduction
. . .checking left irreducibility
. . .checking redundant FDs
Decomposing. . .
. . .checking lossless join property

The following 3NF subschemata
give the dependency preserving
decomposition of your schema:

t_1
    [a,b,c]
    one key: [a]
t_2
    [c,d]
    one key: [c]

The user can then return to the main menu, and either enter another schema, or exit the system. We intend to add more interactive help as we develop the software further (see Section 6).

4. Software Implementation

4.1. Basic Algorithms

The development of increasingly rigorous normal forms, as outlined above in Section 2, has been accompanied by the development of algorithms that are able to perform the normalization process. A number of algorithms exist that, for a given input, can create a full schema with all the constituent relations in a specific normal form. In addition, it is possible to algorithmically test a given relational schema to determine the level of normalization achieved.

The ability to create and evaluate schemas is clearly useful for an educational tool designed to assist students to learn about dependency theory. Students can be shown the normalization process step by step, with help available to hand. They can also undertake any number of exercises, either set by a tutor or of the student's own devizing, and have their results expertly checked. It is this latter capability, in particular, which makes our software a useful learning tool.

In overview, our learning tool includes implementations of the following algorithms. We use Ullman's FD-closure algorithm [22]; we use Beeri and Honeyman's algorithm [1] for checking dependency-preservation after decomposition; we use Loizou and Thanisch's approach [14] for checking for a lossless-join decomposition; we use Ullman [22] for finding a minimal cover for a set of FDs; we use Gottlob's method [12] for computing a cover for the projection of a set of FDs onto a subschema of the decomposition; and we use Luchessi and Osborn's key finding algorithm [15] to identify candidate keys. Several decomposition algorithms are available; we used the proposal in Ullman [22] for 3NF and Tsou and Fischer's approach [21] for BCNF.

4.2. Implementation

The interface program is written in Java. Java has many advantages for this kind of application. It is widely used, it has comprehensive internet support (see below), and it is easy to access applications written in a variety of other languages (through the *Java Native Interface (JNI)* mechanism).

In addition, we use XSB to implement the main logic programs that implement the normalization algorithms. XSB runs on a number of platforms and offers excellent performance that has been demonstrated to be far superior to that of traditional Prolog-based systems [19]. Calls to XSB from the interface program are handled by the YAJXB [7] package. YAJXB makes use of Java's JNI mechanism to invoke methods in the C interface library package supplied by XSB. It also handles all of the data type conversions that are needed when passing data between C and Prolog-based applications. YAJXB effectively provides all the functionality

of the C package within a Java environment.

Although we have used YAJXB in our implementation of composite systems, we note that a number of alternative options exist. Amongst the options that we considered for implementing composite systems were Interprolog , a Java-based Prolog interpreter (e.g., JavaLog), or a Sockets-based, direct communication approach.

Each of the above approaches has its own distinct drawbacks when compared with the approach that we adopted. Interprolog does work with XSB, so we could still take advantage of the latter's performance capabilities. However, Interprolog is primarily a Windows-based application. All of our development was done on a Sun Sparc/Solaris system; YAJXB, though primarily configured for Linux, compiles easily on Solaris. JavaLog was discounted because we felt that it did not offer sufficient flexibility compared with XSB. Finally, using sockets would give us a less portable application because it would involve considerably more application-specific coding. Overall, we felt that the straightforwardness of the YAJXB interface makes it preferable to the Interprolog approach so far as interfacing with XSB is concerned. Moreover, XSB's highly developed status and excellent performance make it more desirable in this context than a Java/Prolog hybrid.

The C library allows the full functionality of XSB to be used. A variety of methods for passing Prolog-style goal clauses to XSB exists. However, we generally found that the string method worked well. This method involves constructing a string σ in a Java String type variable, and using the $xsb\_command\_string$ function (or similar) to pass σ to XSB. This approach allows any string that could be entered as a command when using XSB interactively to be passed to XSB by the interface program. YAJXB creates an interface object; the precise method of doing this is a call like:

```
i=core.xsb_command_string
    (command.toString());
```

where the assignment, as one would expect, handles the returned error code. Variations on this method allow for the return of data where relevant.

Overall, this architecture gives us a great deal of flexibility. Using XSB gives us all the power of Prolog, which has enabled us to use well-documented standard algorithms when constructing the software. Prolog has been widely used in the development of educational tools (see, for example, [2] and [17]). Java would enable this to be further developed as a web-enabled tool. In addition, Java's ability to interface with a DBMS would allow us to incorporate a more sophisticated storage mechanism for retaining multiple sessions, sample databases, and so forth.

5. Evaluating The Software

5.1. Formative Evaluation

For the formative evaluation of our software, we sought comments from several colleagues involved in teaching the material; these were our "expert reviewers" [20]). We additionally used a group of four volunteer students to test the software; these students were learning about dependency theory at the time at which the formative evaluation of the software was being conducted. A number of suggestions made by the expert reviewers and the volunteer students were used to make minor modifications to our initial design.

We then conducted a program of small-group testing, performed over a four week period involving approximately six hours of contact time spread over six sessions, with a set of four student volunteers, 2 from each of the author's institutions. We gathered feedback about the software by using observations and informal "interviews". This involved one of the authors sitting with the students and asking them to articulate their feelings about the learning package as they used it. We made a few further minor modifications as a result of this process.

The students all reported that the software was useful in terms of helping to develop their understanding of dependency theory, and all agreed that the facility for testing their own solutions to normalization problems was motivating to use and important in developing understanding.

5.2. Summative Evaluation

The field test of the software was carried out with a small group of 15 postgraduate students at the University of Westminster who were studying dependency theory as part of a database design module, and took place in the 2002/3 academic year. Students were introduced to the learning tool in a 2 hour tutorial session; they then used the tool for the next three weeks, both during tutorial and independent study time.

At the end of this time, we used a 5-point Likert scale to evaluate the students' perceptions of the software. The Likert scale included a total of 24 statements (with an equal number of positive and negative statements). These items were divided into three categories. Eight of the statements were intended to measure the extent to which our software and a standard text (we used [8]) were perceived as being of value in facilitating student understanding of dependency theory, a further eight items were intended to help to decide the extent to which the software and [8] were motivating to use, and the remaining eight statements were used to collect the students' opinions on the value of comparable features of the software and [8] (i.e., their explanations, exercises and examples). Students were asked to indicate

their strength of agreement/disagreement with each statement in the Likert scale. The five options were: strongly agree, agree, unsure, disagree and strongly disagree.

13 responses to the Likert scale were returned. To produce the measures of student attitudes, a three-stage approach was adopted. The initial step involved "signing" the 24 items included in the Likert scale as being a positive or negative statement about the software and [8]. Next, the returns were analysed using the following system: for each positive statement a response of "strongly agree" was given a score of 5, an "agree" response was given a score of 4, a score of 3 corresponded to an "undecided" response, "disagree" was scored as a 2, and "strongly disagree" was scored as a 1. Conversely, for each negative statement, a "strongly agree" response was given a score of -5, "agree"was scored as -4, "undecided" was recorded as a -3, "disagree" was given a score of -2, and -1 corresponded to a "strongly agree" response. By summing the scores for each return, a figure corresponding to the respondent's attitude towards the software and [8] was computed. In the final step, the [8] score for each respondent was subtracted from the score for the software. This calculation gave a measure of a respondent's attitude to the software that is relative to their attitude towards [8].[1]

To analyse the information produced from the Likert scale, t-statistics were computed to compare the mean scores for the perceptions students had of the software and [8] .

In the overall measure of the two methods, the average difference in the ratings of the software and [8] produced a t-statistic of 2.75 in favour of the software, and no student reported that [8] was "better" than the software. The t-statistic for comparative ratings is significant at the 2% level. The t-statistic for the comparison of average differences was 2.11. This value is statistically significant at the 5% level. Unsurprisingly, given the overall results, the software was also perceived to be "better" than [8] in all three of the sub-categories of Likert scale items.

6. Conclusions and Further Work

The results of our analysis of the software suggest that the tool is of value to students learning about dependency theory. There are several ways in which the tool could be further developed. In particular, we plan to produce a web-based interface, which will make the tool both easier to use, and more widely accessible. In addition, we plan to extend the help information, so that students will have access to pop-up help windows at all stages of the normalization process. We would also like to conduct additional tests of the software, possibly including trials to attempt to determine whether students using the tool are able to demonstrate improved learning by a concomitant improvement in assessment scores.

[1] A positive score indicates a more favourable attitude towards the software than [8]; a negative score represents a more favourable attitude towards [8] than towards the software.

References

[1] C. Beeri and P. Honeyman. Preserving functional dependencies. In *SIAM Journal of Computing*, 10(3), pages 647–656, 1981.

[2] I. Bratko. *PROLOG Programming for Artificial Intelligence*. Addison-Wesley, 1986.

[3] B. Byrne. Top down approaches to database design tend to produce fully normalised designs anyway. In *Proceedings of TLAD*, 2003.

[4] E. Codd. A relational model of data for large shared data banks. In *CACM 26(2)*, pages 120–125, 1970.

[5] E. Codd. *Further Normalization of the Data Base Relational Model*. Prentice-Hall, 1972.

[6] E. Codd. Recent investigations in relational database systems. In *IFIP*, pages 1017–1021, 1974.

[7] S. Decker. *Yet Another Java XSB Bridge*. http://www-db.stanford.edu/%7Estefan/rdf/yajxb/.

[8] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 2001.

[9] R. Fagin. Multivalued dependencies and a new normal form for relational databases. In *TODS 2(3)*, pages 262–278, 1977.

[10] R. Fagin. Normal forms and relational database operators. In *SIGMOD*, pages 153–160, 1979.

[11] R. M. Gagne. *The Conditions of Learning*. Holt, Reinhart and Winston, 1970.

[12] G. Gottlob. Computing covers for embedded functional dependencies. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 58–69, 1987.

[13] W. Kent. A simple guide to five normal forms in relational database theory. In *CACM 13(6)*, pages 377–387, 1970.

[14] G. Loizou and P. Thanisch. Testing a dependency-preserving decomposition for losslessness. In *Information Systems*, 8(1), pages 25–27, 1983.

[15] C. Lucchesi and S. Osborn. Candidate keys for relations. In *Journal of Computer and System Sciences*, 17(2), pages 270–279, 1978.

[16] F. Marton and P. Ramsden. *What does it take to improve learning?* Kogan Page, 1988.

[17] J. Nichol, J. Briggs, and J. Dean. *Prolog, Children and Students*. Kogan-Page, 1988.

[18] J. Rissanen. Independent components of relations. In *TODS 2(4)*, pages 317–325, 1977.

[19] K. Sagonas, T. Swift, and D. Warren. Xsb as an efficient deductive database engine. In *ACM SIGMOD Proceedings*, page 512, 1994.

[20] M. Tessmer. *Planning and Conducting Formative Evaluations*. Kogan-Page, 1993.

[21] D. Tsou and P. Fischer. Decomposition of a relation scheme into boyce-codd normal form. In *SIGACT News 14(3)*, pages 23–29, 1982.

[22] J. Ullman. *Principles of Database and Knowledge-base Systems, Volume I*. Computer Science Press, 1988.