Proceedings of the IASTED International Conference
APPLIED INFORMATICS
Symposium 1, Artificial Intelligence and Applications
Februari 19-22, 2001, Innsbruck, Austria

APPLYING SEMANTIC DATABASE PRINCIPLES IN A RELATIONAL ENVIRONMENT

BEREND DE BOER[1] AND J.H. TER BEKKE[2]
[1]NederWare, Peperstraat 29, 5311 CS Gameren, The Netherlands, berend@acm.org,
[2]Delft University of Technology, Zuidplantsoen 4, 2628 BZ Delft, The Netherlands, j.h.terbekke@its.tudelft.nl,

ABSTRACT

SQL generally allows several solutions to one single problem. The reason for this is that SQL's language concepts lack orthogonality. Because more solutions are allowed, SQL users are uncertain whether the specific solution they chose is correct and performs well. This paper presents a method to tackle this practical problem by introducing the orthogonal semantic concepts from the Xplain data language in the development phase of relational databases. This conversion method results in unique, consistent, well-performing and portable SQL definitions and SQL queries. Over a period of several years, we tested the approach extensively and applied it in practice, for which we used databases and queries of different complexities. The method proved to be time saving in both design and implementation. This paper presents an overview of the principles used in the automated tool for converting safe Xplain into equally safe SQL.

Keywords: intelligent databases, query languages, data models.

1. INTRODUCTION

The basic constructs of SQL are easy to learn. As users move on to more advanced queries like joins or correlated sub-queries, however the lack of orthogonality and the abundance of pitfalls [3], [4], [10] can cause confusion. These days there is a practical alternative, one that can even be used in more phases of building information systems. We advocate the use of Xplain [9], which with the advent of a new tool that can create SQL from Xplain statements has become quite practical. Xplain has both a visual and textual component, as will be shown in the next section. Both its visual and textual presentation are extremely simple and can be explained in a matter of minutes to any audience. Xplain does not offer the multiplicity of relationships found in the information modeling literature, but offers all possible cases with these two relationships: aggregation and specialization (including multiple inheritance) [7]. The visual side of Xplain does not strain people's cognitive abilities. It just offers easy drawing and distinction of aggregation and inheritance. It is practical to use Xplain when a certain exactness has to be introduced during design or requirements analysis phases. The visual model can be refined from high-level business semantics to the semantic detail necessary for information systems. The visual design can be coded straightforwardly into the language of Xplain. The tool presented in this paper, Xplain2sql, provides an additional incentive to capture knowledge into semantic database models, because this tool can transform these Xplain statements into ANSI SQL and its many dialects. Both data definition and data manipulation are supported. The transformation process has built-in expert knowledge about the targeted SQL dialect to produce optimal SQL code. This paper is organized as follows. After a short introduction on the semantic Xplain concepts, an overview is given of the techniques used for converting semantic definition and manipulation commands into SQL equivalents. Also some insight in the implementation is given. The last section includes an overview of the history of the automated tool and gives information about some practical application areas in which the system has been used successfully.

2. XPLAIN CONCEPTS

In designing databases, a conceptual model is produced, which consists of descriptions of relevant object types. Abstraction is a vital concept in semantic databases; three types are distinguished: classification, aggregation and generalization. They make use of the fundamental *type-attribute* relationship.

2.1 *Classification*

The real world is described by considering the properties of relevant objects, where a property is defined as a fundamental notion, but no value is assigned to it. The abstraction leading to a property is called classification. The examples (i.e. instances) occurring in a database and required for the recognition of a property are purely meant as illustrations. The property is not being defined hereby. Properties are represented by rectangles in diagrams, see figure 1 and figure 2. The counterpart of classification is called instantiation.

2.2 *Aggregation*

Aggregation is defined as the collection of a certain number of properties in a type, which in itself can be regarded as a new property (note the analogy with the mathematic.Ial set concept). A property occurring in an aggregation is called an attribute of the type. An empty type is called base. Aggregation allows view independence: we can discuss the obtained type (possibly as a property) without referring to the underlying attributes. By applying this principle repeatedly, we can set up a hierarchy (in the sense of a network) of properties. Examples are given in figure 1 and figure 2. Normally the hierarchy contains only aggregated types. Aggregation is indicated by a line connecting the centers of two facing rectangle sides, while the aggregate type is (according its definition) placed above its attributes. Of course, aggregation also has its counterpart: the description of a type as a set of certain attributes is called decomposition. We define a type by listing its attributes, so we could have the following type definitions:

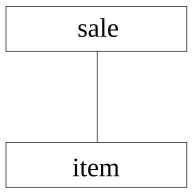

Figure 1. Aggregation

type item (A4) = description, stock, price.
type sale (A4) = week, day, item, number, amount.

Type definitions carry semantics; they contain the essential properties and relation- ships. Aggregations can be described by means of the verb to have. According to the above type definition, an item has a description, stock and price. Identifications are properties denoted by type names. This interpretation implies singular identifications. Attributes (not types!) may contain roles. An example is ‘item description’ related to type ‘description’. Roles are separated from the type by an underscore.

2.3 *Generalization*

The third type of abstraction, important to conceptual models, is generalization; it is defined here recognizing of similar attributes from various types and combining these in a new type (note the analogy with the intersection operation from mathematical set theory). We can equally discuss the new type without mentioning the underlying attributes, and it can in itself again serve as a property (i.e. it allows view independence). An example is adding the type ‘invoiced sale’ to the foregoing definitions. This type has all the attributes of type sale and additionally the attributes: name, street, zipcode and city. The type ‘invoiced sale’ must therefore be conceived of as the specialization of the type sale. In abstraction hierarchies, generalizations are schematically represented by a line that connects facing corners of rectangles, where the generalized type is placed below the specialized ones. The opposite of generalization is called specialization. The generalization together with the attributes to be added to it is (by definition of the concept) described in the type definition of the specialization. So the type definition of the specialization of sale is:

type invoiced sale (A4) = [sale], name, street, zipcode, city.

Specializations are commonly associated with the verb to be. According to the above type definitions, an invoiced sale is a sale with name, street, zipcode and city.

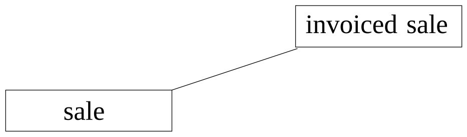

Figure 2. Generalization

3. CONVERSION OF DEFINITIONS

The Xplain language has several commands for data definition and data integrity. This section shows the conversion of the base, type and init commands to SQL. Our first complete semantic example (including base types) is as follows:

base day (A3) ("Mon","Tue","Wed","Thu","Fri","Sat").
base week (I2).
base amount (R4,2).
base description (A13).
base stock (I4).
base price (R4,2).
base number (I4).
base name (A60).
base street (A60).
base zipcode (A10).
base city (A40).
type item (A4) = description, stock, price.
type sale (A4) = week, day, item, number, amount.
type invoiced sale (A4) = [sale], name, street, zipcode, city.

3.1 *Converting base commands*

Depending on the capabilities of the target SQL implementation, the base command is converted to a UDT (User Defined Type) or not converted at all. When it is not converted, the domain is simply remembered and used wherever a base definition occ taturs in a type definition. An example of this is shown in section 3.2. For example, the conversion by Xplain2sql of the foregoing base definitions into the InterBase SQL implementation results in the following domain definitions:

```
create domain Tday as character(3) not null
check (value in ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'));
create domain Tweek as smallint;
create domain Tamount as float;
create domain Tdescription as character(13);
create domain Tstock as smallint;
create domain Tprice as float;
create domain Tnumber as smallint;
```

InterBase SQL also supports domain restrictions, see the definition of domain Tday. The domain restriction output is suppressed if the target SQL implementation does not support them.

3.2 *Converting type commands*

The type command is converted into the SQL create table statement. For example, the conversion of the above-listed type definitions (including domains) to InterBase results in the following SQL definitions:

```
create table item (
    id_item character(4) not null primary key,
    description Tdescription not null,
    stock Tstock not null,
    price Tprice not null);

create table sale (
    id_sale character(4) not null primary key,
    week Tweek not null,
    day Tday not null,
    id_item character(4) not null
    constraint cst_sale_1item references item (id_item),
    number Tnumber not null,
    amount Tamount not null);
```

The type conversion process handles the following three aspects:

1. The external representation of the type identification is converted into a standard column with a corresponding representation: either integer or character. This column has the primary key constraint. If the type identification representation is integer, Xplain2sql attempts to create an auto-incrementing primary key if the target SQL implementation supports it and if this option has not been disabled by the user.
2. There are several semantic arguments against null values. Commonly a specializa- tion is the most appropriate modeling solution. That is why Xplain does not support these values. According to the Xplain manual [12], every attribute of a type should therefore have a value. And as Xplain does not have the concept of nulls, therefore every column should be not null. Xplain supports specification of possibly derived default values, therefore a user does not need to specify a value for every column in his insert statements.
   In a relational context we can also discover several arguments against nulls. This helps us, for example, to avoid unpleasant surprises in where clauses, and to avoid situations where indexes cannot be defined.
   When the UDT is being created, most SQL implementations allow specification of the default null/not null option for that type in the create table statement. For reasons of clarity, Xplain2sql therefore writes always not null or null in the create table statement.
   There are two exceptions: one exception is explained in section 3.4. The other is when columns contain binary large object (BLOB) data (memo or graphic representations). These columns are null by default. Even when the empty string is assigned to a memo field, up to one database page is used (2K or even more). So using null in such cases makes sense.
   There is one more exception: a user can force a nullable attribute, base or type, by putting the Xplain2sql specific keyword optional in front of it. This is useful when the datamodel is reverse engineered from a relational model.
3. If an attribute refers to another type, Xplain2sql adds the foreign key key constraint to that table.

Xplain2sql also recognises the semantic abstraction of specialization. Consider the specialization in foregoing semantic model:

```
type invoiced sale (A4) = [sale], name, street, zipcode,
     city.
```

This is converted to:

```
create table invoiced sale (
    id_invoiced_sale character(4) not null primary key,
    id_sale character(4) not null unique
    constraint cst_invoiced_sale_1sale
        references sale (id_sale),
    name Tname not null,
    street Tstreet not null,
    zip_code Tzip_code not null,
    city Tcity not null);
```

Specializations are converted into 1-to-(0,1) relationships by means of a foreign key and a unique constraint. So every sale can occur at most once as invoiced sale. Note here that Xplain supports multiple inheritance, so using the parent key as the subtype key will not work.

3.3 *Naming conventions*

Converting Xplain identifiers (i.e. base names or type names) into SQL identifiers is not always straightforward. The first reason is that Xplain supports spaces in identifiers. The second is that an Xplain identifier might be a reserved word in the target SQL implementation. Most SQL implementations support so-called quoted identifiers, which are part of the ANSI-92 standard. Identifiers surrounded with double quotes can contain spaces or can be reserved keywords. The example below shows the conversion of the type item to PostgreSQL. This dialect supports quoted identifiers, but it does not support UDTs.

```
create table "item" (
      "id item" character(4) not null primary key,
      "description" character(13) not null,
      "stock" int2 not null,
      "price" numeric(6,2) not null);
```

Quoted identifiers do not solve all identifier problems. For example, UDTs usually cannot be quoted. So column names can be quoted, but not their data types. For base name conversion another technique is needed: replace the spaces in base names by underscores (‘_’) and add an underscore as suffix or prefix if the resulting identifier is a keyword in a particular SQL implementation. Microsoft SQL Server 7.0 supports quoted identifiers, but also bracketed identifiers: identifiers surrounded with square brackets. Square brackets seem to be allowed everywhere, even in column types, so base types can have spaces too in this implementation.

3.4 *Converting init constraints*

The init statement is one of the most difficult statements to convert. ANSI-92 SQL supports something comparable to init in the form of the keyword default. But only literal expressions (i.e. numbers or characters) are allowed. As an example we initialise the number of every sale. This default can be overridden when a new sale is inserted, but if no number is specified, the number will be 1:

```
init default sale its number = 1.
```

As this is a literal expression, conversion can be done as follows:

```
create table sale (
      id_sale character(4) not null primary key,
      week Tweek not null,
      day Tday not null,
      id item character(4) not null
      constraint cst_sale_1item references item (id_item),
      number Tnumber, default 1 not null,
      amount Tamount not null);
```

An init expression that is not a literal is much harder to support. The target SQL im- plementation should have the capability to specify a before-insert trigger for the init default command and an after-insert trigger for the init command. For implementations not supporting before-insert triggers, columns with inits must allow null, so it is allowed not to specify a value. Triggers are usually fired after constraint checking. So a not null specification requires a value to be specified, even if an init expression is specified. In the trigger itself the value of the column will be replaced if it is an init expression or conditionally replaced if it is an init default expression. As an example, consider the following type definitions:

```
type item (A4) = description, stock, price.
type sale (A4) = week, day, item, number, unit price,
       amount.
init sale its unit price = item its price.
```

The Microsoft Transact SQL code becomes:

```
create trigger [tr saleInit] on [sale]
for insert
      as update[sale]
      set
            unit price= [item].[price]
      from inserted
      join [sale] on [sale].[id sale] = inserted.[id sale]
      join [item] on [item].[id item] = [sale].[id item]
go
```

4. CONVERSION OF QUERIES

The Xplain language has several commands to retrieve and modify data. This section presents conversion for the retrieval commands get and extend. The data modification commands insert, update and delete are trivial. Currently, Xplain2sql does not support the new (recursive) cascade command.

4.1 *Converting get commands*

Obviously, the get statement is converted into a select statement. For example, the Xplain command to list all items is:

```
get item.
```

Converted into (InterBase) SQL this becomes:

```
select *
from item;
```

If the its construct occurs, a join is generated; this is a save conversion. The Xplain command to get all sales, including item descriptions, is:

```
get sale its week, day, item its description.
```

Converted into (InterBase) SQL this becomes:

```
select sale.id sale, sale.week, sale.day, item.description
from sale
join item on item.id item = sale.id item;
```

Not every SQL implementation supports the above ANSI-92 style joins. For implementations that do not support the new-style join, the old-style join is generated. PostgreSQL is an example:

```
select "sale"."id sale", "sale"."week", "sale"."day",
      "item"."description"
from "sale", "item"
where ("item"."id item" = "sale"."id item");
```

4.2 *Converting extend commands*

The Xplain extend command adds a new temporary column to a table. This column is used for intermediate results in complex derivations. No SQL implementation known by the authors supports temporary columns. Only some of the implementations support temporary tables, which are almost as useful. It is possible to create a new, temporary table with two columns: one contains the primary key of every column from the table which is extended; the second contains the contents of the new column. A join of these two tables has the effect of a table extended with an additional column. As an example we determine the turnover per item. In Xplain this is formulated as follows:

```
extend item with turnover =
    total sale its amount
    per item.
```

```
get item its turnover.
```

Converted into Microsoft Transact SQL, this becomes:

```
select
    [id item],
    (coalesce(
        (select sum([sale].[amount])
            from [sale]
            where
            ( [sale].[id item] = [access type].[id item] )
    ), 0)) as [turnover]
    into [#item.turnover]
from [item] access type
```

```
select [item].[id item], [turnover]
from [item]
    join [#item.turnover] on
        [#item.turnover].[id item] = [item].[id item]
```

The extend command is converted into a select, which does not output the results, but inserts them in a table because of the into clause. This table is temporary because of the '#' sign in front of the table name. The second select just joins the original table with the temporary table to output the results. The first select statement uses a correlated subquery to get the sum of sales for every item. It is important that the SQL pitfall of missing items can be avoided [4, 10] by selecting from the item table. Because of missing sales, certain sums can be null. The nulls are another pitfall, because a user expects to be able to select items with no sales with something like 'turnover = 0'. Using the IsNull function we insert a 0 instead of a null if a sum is null. Using this example we can now define three requirements an SQL implementation has to support to be able to correctly convert an extend command without pitfalls:

1. The implementation should have the notion of a temporary table;
2. The implementation should support subqueries;
3. The implementation should have a function like coalesce (ANSI-92 standard). This function is also called IsNull or NVL in some implementations.

The authors only know two two SQL implementations which fulfil these three requirements: Oracle and Microsoft SQL Server. For implementations that do not support the last two requirements, Xplain2sql cannot provide a conversion. Users have to revert to the tedious manual conversions. For implementations only lacking the notation of a temporary table, some alternatives can be given:

- Use a view instead of a temporary table. However, it is possible that not every user is authorized to create views. Therefore this approach probably works only for static cases, where the possible extends are preformulated. Views created on the fly by one user are visible for and usable by others users of the same database. To avoid naming conflicts, view names have to be unique to allow on the fly view creation to work.
- Use a normal table instead of a temporary table. Normal tables are also subject to the problems listed above.
- In-line the extend whenever the extend occurs in a select. The extend command is not translated in this case, but simply remembered. When the outcome of an extend is used in a get command, the subquery is written at that place. Unfortunately, this can lead to awful and inefficient queries.

5. IMPLEMENTATION ASPECTS

Xplain2sql is an Open Source program written in Eiffel. Using the SmallEiffel compiler [2], Xplain2sql can be made available on 22 platforms. Xplain2sql won a silver award at the Eiffel '99 Struggle. Xplain2sql is a fine example of the Builder pattern [5], [6]. The Director participant is implemented by the tokenizer and parser in Xplain_scanner and Xplain_parser. The Builder participant is implemented by the class SQL_GENERATOR. For expression parsing the Interpreter pattern [5], [6] is used. More information about the implementation can be found in the manual [1]. Full source, documentation, samples and binaries for FreeBSD, BeOS and Microsoft Windows NT are available at http://www.pobox.com/berend/ xplain).

6. SUPPORTED SQL DIALECTS

Currently Xplain2sql converts Xplain to Microsoft SQL Server, Inprise Interbase, PostgreSQL, ANSI-92 SQL, and others. It must be noted here that Xplain generates more than just create table statements; it also generates insert, update and delete stored procedures for each table. The latest version can generate ADO middleware code for Delphi. Besides the advantage of a language that supports both aggregation and specialization with ease (unlike SQL), porting database code comes almost for free.

7. PRACTICAL EXPERIENCES

Xplain modeling has been used throughout NederWare's consultancy practice for almost 7 years. But the manual conversion to SQL proved to be tedious. In 1996 a precursor of Xplain2sql was used to develop and generate the database of an administrative program of the Urk Fish Auction, the largest flatfish auction in Europe [8]. This database consisted of more than 110 tables. Some smaller examples where Xplain2sql has been used to design and generate the database include a book loan and library management program (Library Gameren), a contact, task and order program (FunderingsTechniek Noord, Tolbert), and a money loan program (Reac, Rotterdam). Currently Xplain2sql is used to (re-)write the core of Ortec's computer-aided planning software for (international) transport. Using Xplain, mapping objects defined in the OO-modeling stage to a relational database proved to be quite straightforward. In all these cases the automatic conversion of Xplain to SQL has proved to be very time-saving: Xplain is not only more compact, but it is also less easy to create design faults. We found that when something could not be expressed in Xplain, the database design could be improved and this improvement would also fit in Xplain. A comparative study of students who designed a relational and semantic database points in the same direction [11].

CONCLUSION

With Xplain2sql it has become practical to use the Xplain language in relational environments. Because Xplain is a concise language that is orthogonal, it is easy to learn. The fundamental is-a and has-a abstractions have elegant language definitions. Definition and manipulation in a database environment therefore become simple activities. The build-in expert knowledge used in the conversion from Xplain to SQL brings these advantages to todays business applications.

REFERENCES

[1] B. de Boer, Xplain2sql manual, URL: http://www.pobox.com/berend/xplain, version 0.7, October 1999

[2] D. Colnet and O. Zendra, Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler, URL: http://SmallEiffel.loria.fr/papers/tools-europe-99.pdf, *Proc. 29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, IEEE Computer Society, (1999), 341-350

[3] C.J. Date, A Critique of the SQL database language. *ACM SIGMOD Record*, 14, 3 (1984), 8-52

[4] C.J. Date, *Introduction to database system (7th edition)*, Addison-Wesley, Reading Mass. (2000)

[5] Eric Gamma et al., *Design Patterns*, Addison-Wesley, Reading Mass. (1995)

[6] Jean-Marc Jézéquel et al., *Design Patterns and Contracts*, Addison-Wesley, Reading Mass. (2000)

[7] B.M. Meyer, *Object-oriented software construction (2nd edition)*, Prentice-Hall (1997)

[8] N. v.d. Laan, *Case Study Visafslag Urk*, URL: http://www.ascbenelux.nl/Projecten/Urk.htm, ASC Software Technologies (1999)

[9] J.H. ter Bekke, *Semantic Data Modeling*, Prentice Hall (1992)

[10] J.H. ter Bekke, Can we rely on SQL?, *Proc. 8th Int. DEXA Workshop '97*, Toulouse France, ed. R.R. Wagner, IEEE Computer Society (1997), 378-383

[11] J.H. ter Bekke, Comparative study of four data modeling approaches, *Proc. 2nd Int. EMMSAD workshop*, Barcelona, eds K. Siau, Y. Wand and J. Parsons, B1-B12 (1997).

[12] J.H. ter Bekke, *Manual Xplain DBMS*, URL: http://is.twi.tudelft.nl/dbs/xplain/manual.html, version 5.8, October 1999 (in Dutch)