Storing Data of Ontology-Based Dynamic Applications

Samuil Nikolov

Eurorisk Systems Ltd.
31, General Kiselov Str., 9002 Varna, Bulgaria

***Abstract**: The publication describes the approach chosen to store ontological data as part of a commercial framework that generates dynamic applications. The approach is flexible, allowing on-the-fly changes of the stored ontology attributes and adding new classes without restructuring the database and even without interrupting user's operation. It is based on multilayer vertical tables and a search table that allows executing fast queries on frequently used parameters.*

***Keywords:** Ontology, Database logical design, Applications with Dynamic User Interface*

1. INTRODUCTION

The goal of the publication is to present the chosen approach to building a database that stores ontological data. In [8], a framework is described that can dynamically generate applications using a set of scripts (models). Each script defines dynamic user interface and represents an ontological class with all its attributes. The relations between classes are represented as a special type of attributes. A set of classes forms a domain which is actually the application generated by the presented framework. This publication aims at sharing experience in designing the database storage for the framework. As the software was initially designed for building banking applications, there were specific requirements that influenced the decisions on database design, as follows:

- Support of historization (versioning) for the stored ontology instances;
- Storing information about the users that created, modified and eventually marked as deleted a specific ontology instance and when these actions occurred;
- Separating the data into different organizational entities, hiding the information between them;
- Possibility to transform stored objects in run time if the attributes of the classes in the system are changed by supplying newer versions;
- Adding new or changing existing parts of the software unnoticeably to the users while impacting as little as possible the operation of the system;
- Support for outside reporting tools like crystal reports.

There are two mainstream approaches to solving this problem – storing the data in object databases that store the ontology class instances automatically or designing a specialized relational database that can store ontological data. The most prominent object databases are db4o [12] and DTS/S1 [13] but they have still not asserted their use - fulness in the world dominated by the relational database model.
The relational databases are an industry standard and a lot of customers already have such database management systems installed and in operation. This makes their usage a preferred solution for commercial software, as it can easily fit into customers' universe.

There are numerous ways to store ontological data inside relational databases. The most common are:

- horizontal class (or table per class approach) [2,3,4,10];
- table per property (attribute) approach [5,6,7];
- hybrid approach from [9];
- vertical table (edge) approach with triplets [1,4].

Some more considerations on the stored data will be introduced besides the above-mentioned requirements in order to clarify the advantages of the chosen approach:

- The stored data consist of relatively few classes (in the vicinity of hundreds), containing lots of different properties, including multi field and tabular ones;
- The system has to store large tabular data as part of a certain multi field attribute of a class. Sometimes the tables stored inside the class instances are very large. For instance, keeping information about all the loans belonging to a certain portfolio of a bank results in storing thousands of loan object identifiers in the portfolio object to account for the existing connections to the loan objects.

2. OVERVIEW OF THE SOLUTION

Considering the mentioned limitations, some of the options of storing the frame- work's ontological data can be ruled out immediately. The table per property and the hybrid approach from [9] are unsuitable due to the vast amount of various properties in- side the system that would make reconstructing an object irrationally difficult [4] and managing the database almost impossible.

The table per class approach is the usual way of creating specialized application databases but it does not suit the requirements for keeping older object versions, as the new versions would transform the table belonging to the current class. Also, it would be difficult to restructure the database by e.g. adding new columns, deleting old ones and creating new tables. This contradicts to the requirement for quick adjustment and least system wide impact when making changes in the system. Besides, some customers will not allow running the system with alter and delete table rights granted to it. The author's experience with building banking software which uses the table per class approach shows that the system has to be as adjustable as possible. Otherwise its support be- comes very difficult and extensibility and operation issues often arise. There are appli- cations (e.g. SAP) that store data across a large number of tables. However, having thousands of tables instead of one makes the system harder to manage and operate [1].

The last option to consider is the vertical table, a widely used way of representing ontological data [1]. The proposed approach is based on it and uses four tables three of which are vertical and the other one holds additional information to facilitate implementation-specific requirements. The vertical table approach allows for extensibility and easy structural changes and is also suitable for managing large classes with many properties. Figure 1 shows an overview of the used database structure. The table RF_HEAD is used as a main search table to find an object's identifier using the most common search parameters. The table RF_BODY is vertical and contains the main attributes of the stored objects. The other two tables, which are vertical as well, are in- dexed using the found object's identifier (SESSION_ID) and a specific attribute of the object (VAR_ID). They contain specific additional data about the attribute in the form of tabular data and metadata about it. The schema will be explained in detail in the following chapters, showing how it solves the previously stated framework specific problems.

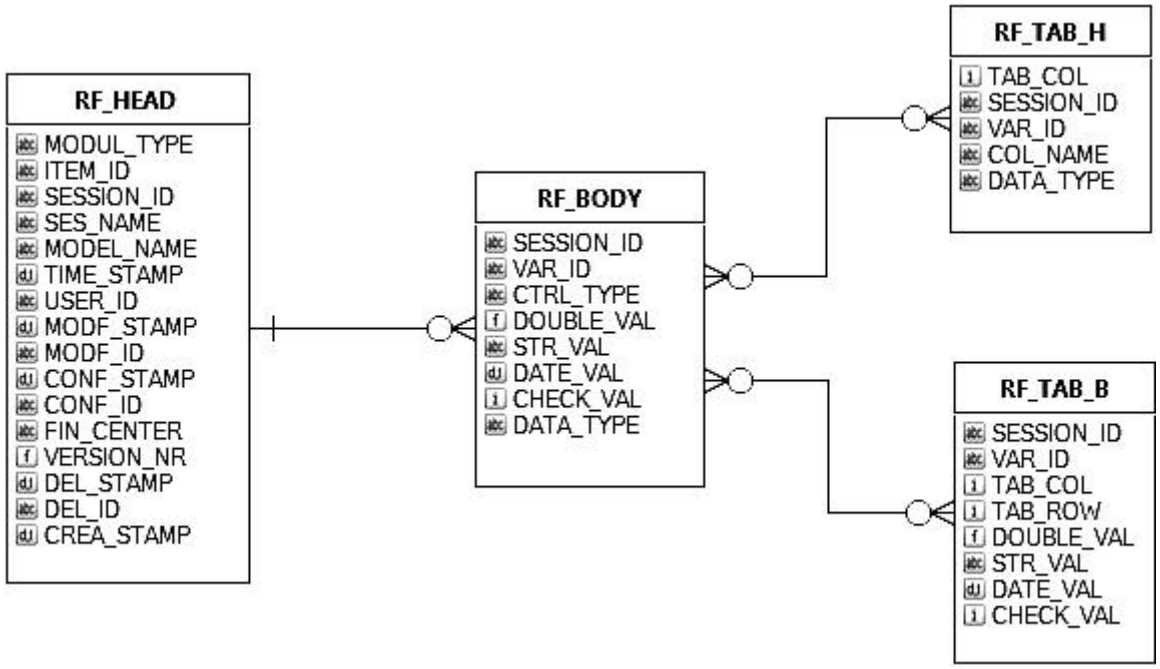

Fig.1. Overview of the used database schema

3. SOLUTION DESCRIPTION

3.1. Representation of ontology class inheritance

In [8] the notion of item identifiers was introduced. An item identifier can have zero or more sessions associated with it, containing stored object data. Item identifiers can be used for semantic grouping of unrelated sessions. Each session has its associated model - the object's ontology class. The sessions themselves cannot be inherited, but an item identifier can have multiple object data associated with it, thus providing an extensibility feature for the data associated to the item identifier. The drawback of the solution is the possible occurrence of inconsistency in case the base class session is omitted and only the subclass session exists.

Figure 2 shows a diagram showing the item identifier "Peter Johnson" with several sessions (or stored class instances) associated with it in two different domains (or framework-generated applications) – the first one in the context of a factory information system and the second one - of a tourist information system. Both have general personal information represented by a saved object (session) of the "Person Data" class. The domain-specific information is represented by the sessions of "Employee Da- ta" and "Customer Data" classes. Also, several current condition data sessions are shown on the figure as different versions of object instances of classes "Last Month Production Data" and "Current Reservation Data". The strength of the proposed incremental approach in building data hierarchies is evident, as in a classical inheritance approach the general and domain-specific data would be duplicated in all the individual current condition objects. The mentioned drawback is also clearly demonstrated – the current reservation data is irrelevant without the personal information or the customer data. Each session object has an identifier constructed from the item identifier and a system-unique number generated in a standard way [11]. This session identifier is used as a part of the primary key in all the tables in the proposed schema.

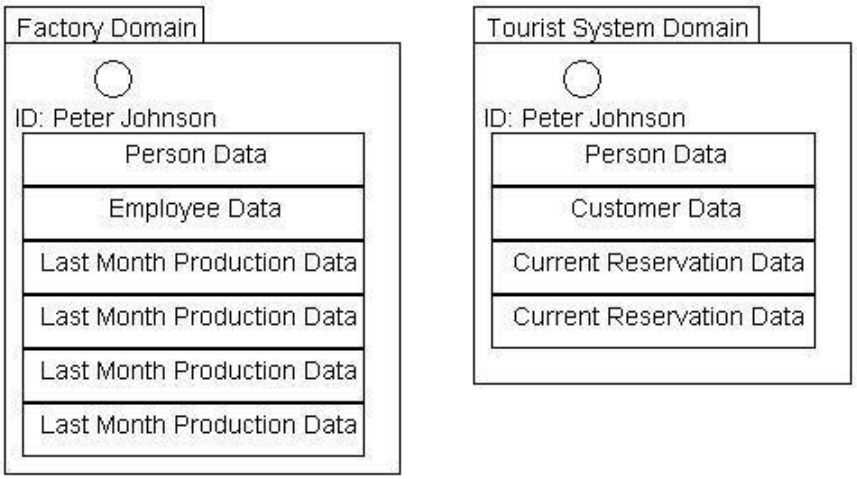

Fig.2. Item identifiers with different sessions in two different domains

3.2. Support of historization, object states and solving data access conflicts.

As mentioned above, the applications generated by the framework have to support object versioning. The need for this is obvious on Fig. 2, where different versions of objects of the "Last Month Production Data" class have to store data about the employee's monthly work at the factory. Also, these have to contain information about who created the specific object data and when, who was the last to modify it and also if this is the final version of the object data (if it is confirmed by a supervising manager). All this information is stored in the main table of our schema – RF_HEAD. It allows searching for the session identifier of an object using information about the object's organizational entity (table column FIN_CENTER), subdomain (table column MODUL_TYPE), item identifier (table column ITEM_ID), class (table column MODEL_NAME), time of creation (table column TIME_STAMP) etc. The purpose of separating frequently used object attributes in a table is to support fast finding of object keys (session identifiers) and to use them to reconstruct session objects, most parts of which are stored in the very large vertical ta- ble RF_BODY.

FIN_CENTER	MODUL_TYPE	ITEM_ID	MODEL_NAME	TIME_STA...	USER_ID	SESSION_ID
001006	Period_Data	Basis Rate FTP	Analytical Schemes	2010-01-25	User 007	Basis Rate FTP@34488
001006	Structure_Data	Market Rate Structure	Structure Definition	2010-01-25	User 007	Market Rate Structure@34493
001010001	OR_STA_Data	001010001	Standardized Approach (STA)	2010-01-26	Admin	001010001@34525
001006	Instrument_Data	FixedAnnuityId5	Cashflow Instrument	2010-07-14	admin	FixedAnnuityId5@116793
001006	Instrument_Data	FixedAnnuityId6	Cashflow Instrument	2010-07-14	admin	FixedAnnuityId6@116796
001006	Instrument_Data	FixedAnnuityId7	Cashflow Instrument	2010-07-14	admin	FixedAnnuityId7@116799

Fig. 3 Main columns of RF_HEAD table in the proposed database schema

The table RF_HEAD also stores additional information about the actions on the object (like deletion and modification time stamps), as well as a version number that ensures data consistency in multiuser operation mode. The number is needed as the time between opening a session in the framework for modification and storing it back to the database can be substantial. Many writes can occur during this time and comparing the initially read version number to the one during writing ensures no overwriting other users' data occurs.

After the required session identifier(s) are found, the sessions can be reconstructed from the data in the table RF_BODY. It is an inlined [6] vertical table meaning that there are columns for all the basic types of attributes. Also, it contains a field specifying the attribute's actual valid type. The strength of the approach is evident in a system that has to be translated into many different languages and also reported by an external tool. If only the strings for multiple choice fields like combo boxes are stored in the database, translating the class description will cause inconsistency that will result in losing the stored currently selected item. If only the indexes are stored, proper reporting will be difficult as the meaning of the values of combo box lists are not always available. There are special cases of multi field attribute types that are linked to the other tables in the schema and solve the remaining problems stated at the beginning of the publication.

3.3. Storing large tabular data as attributes

The dynamically generated user interface in the framework from [8] contains grids that can have dozens of columns and thousands of rows of data. If these were stored in the common attribute data table – RF_BODY, it would grow quickly to be enormous. That's why we introduce a separate table to store them. It is indexed on the object's session identifier and the attribute identifier (VAR_ID) of the grid. The grids contain class-specific multi field attribute data and in special use cases connect the current object to objects of other classes in the system by storing their item identifiers. Thus, the relations between the ontology objects are accomplished. The table that stores the grid data - RF_TAB_B is also a vertical inlined table that contains cell descriptions of the grid data in the form of triplets (row, column, value) but this time indexed not only on the session identifier, but also on the attribute identifier.

3.4. Support for external reporting

In order to generate meaningful reports from the data stored in the framework, one has to obtain some metadata about the attributes of the class objects. In case of simple attributes, the attribute name, type and string representation are usually sufficient but when storing the large tabular data described in the above paragraph one will also need information about the name and type of each column in the stored table. This is solved using another vertical table – RF_TAB_H - storing the column number, name and type for a certain tabular attribute in the class description (Fig. 1).

3.5. Support of class attribute changes in run time

The class descriptions, represented by model scripts in the dynamic application framework can be modified to replace the existing ones in a user's installation while the framework is operational. Therefore, the data stored for a certain object should be transformed, on user request, taking into account the new attribute descriptions. In this process, two phases should be distinguished:

- Determining if the associated model has been changed when a session is opened for modification or examination;
- Mapping stored data to the attributes described in the model.

In the first phase the stored attribute identifiers and their types are used to deter- mine if any changes have occurred. The table descriptions are mapped according to the column types, stored in RF_TAB_H table.

In the second phase, the attributes with matching identifiers in the class description and in the database, are retained while the new ones in the class are added with default values according to their type. Attributes in the database that have no corresponding at tributes in the model are removed. This is done only on user request. In some cases, the old data in the database could be retained for historization. For grid data, the columns that have the same type are retained and the columns that have different data type or are newly added are given default values. If the grid description in the class contains fewer columns, the RF_TAB_H and RF_TAB_B table data is trimmed to the specified size removing the last columns.

4. CONCLUSIONS

A specific solution for storing data of ontology-based dynamic applications is pro- posed in the publication. The introduction of the main search table allows quick reconstruction of ontology objects using frequently queried parameters. Besides, it is suitable to store additional specific information about the ontology objects like version, users that created and modified it, when that occurred etc. The multilayer vertical table organization makes possible easy changes in the existing parts of the software while impacting the system as little as possible. Also, it permits storing of large grids' data and metadata about them. This approach allows substituting class descriptions in run time by transforming sessions and the usage of external reporting tools. The multilayer organization can be extended with tables storing other specific data like BLOBS and CLOBS if these are necessary.
The database organization allows the flexibility and extensibility of the dynamic application framework presented in [8].

5. REFERENCES

[1] Agrawal, R., Somani, A, Xu, Y. (2001) Storage and querying of e-commerce data, *VLDB. Morgan Kaufmann*.

[2] Astrova, I., Nahum, K., Ahto, K. (2007) Storing OWL Ontologies in SQL Relational Databases, *Engineering and Technology* 1.4, 167-172.

[3] Chen, Li, Martone, M., Gupta, A., Fong, L., Wong-Barnum, M. (2006) OntoQuest: exploring ontological data made easy, *Proc. of the 32nd Int. Conf. on VLDB*.

[4] Dehainsala, H., Pierra, G., Bellatreche, L. (2006) Managing instance data in ontology-based databases *Technical report, LISI-ENSMA.*

[5] Florescu, D., Kossmann, D (1999) A performance evaluation of alternative mapping schemes for storing XML data in a relational database *Tech. Rep., INRIA.*

[6] Florescu, D., Kossmann., D. (1999) Storing and Querying XML Data using an RDMBS *IEEE Data Engineering, Bulletin* 22(3), 27-34.

[7] Gali, A., Chen, C., Claypool, K., Uceda-Sosa, R. (2004) From Ontology to Relational Databases *In Proc. of ER (Workshops)*, 278-289.

[8] Nikolov, S., Antonov, A. (2010) Framework for building ontology-based dynamic applications *ACM Int. Conf. Proc. Series (ICPS)* Vol. 471, 83-88.

[9] Pan, Z., Heflin, J. (2004) DLDB: Extending Relational Databases to Support Semantic Web Queries *Design*, 303-308.

[10] Vysniauskas, E., Nemuraite, L. (2006) Transforming ontology representation from OWL to relational database *Information Technology and Control* 35.3A.

[11] Wieringa, R., de Jonge, W. (1991) The Identification of Objects and Roles Amsterdam *Technical Report IR-267*.

[12] db4Objects - http://www.db4o.com/

[13] DTS/S1 'Pitch Black' - http://www.obsidiandynamics.com/dts/index.html